AN ABSTRACT OF THE THESIS OF

Muhammed Saleh Al-Mulhem for the degree of Doctor of Philosophy in Computer Science presented on December 14, 1989.

Title: DataLab: A Graphical System for Specifying and Synthesizing Abstract Data Types.

Redacted for Privacy

Abstract approved: _____

Ted G. Lewis

Formal methods using text to specify abstract data types (ADTs) are powerful, but they require great effort and a high level of expertise. Visual programming languages present an alternative way of programming but are limited to building small programs. This research presents an approach for specifying ADTs using a combination of text and visual objects. Furthermore, it presents two algorithms to map those specifications into imperative code. DataLab, a computer program for the Macintosh™ computer, is an implementation model for this approach.

DataLab consists of two major components: a graphical editor and a source code generator. The graphical editor allows the user to build a specification consisting of an interface part and an implementation part for each ADT. The interface of the ADT is

specified textually in a window that is part of the graphical editor. The implementation part of the ADT includes the operations, which are specified in DataLab as a set of "Condition/Action" transformations. These transformations describe the behavior of the operations and are built by selecting graphical objects from a palette and placing them on the screen. The source code generator takes the specification of the ADT as an input and generates an encapsulated Pascal code. It consists of two algorithms: the first maps the specification into its semantics, and the second maps the semantics into Pascal modules.

DataLab: A Graphical System for Specifying and Synthesizing Abstract Data Types

by

Muhammed Saleh Al-Mulhem

A THESIS

submitted to

Oregon State University

in partial fulfillment of

the requirements for the

degree of

Doctor of Philosophy

Completed December 14, 1989

commencement June 1990

APPROVED:

Redacted for Privacy

Professor of Computer Science in Charge of Major

Redacted for Privacy

Head of Department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis presented December 14, 1989

Typed by Muhammed Saleh Al-Mulhem for Muhammed Saleh Al-Mulhem

ACKNOWLEDGMENTS

I would like to express my true appreciation to my advisor Prof. Ted Lewis, for his guidance, encouragement, and helpful discussions. It has truly been a privilege to work with him.

I would like to thank my committee members: Prof. Tim Budd, Prof. Bruce D'Ambrosio, Prof. Toshi Minoura, and Prof. Walter Loveland for their helpful comments.

I would like to thank King Fahd University of Petroleum and Minerals, and specially Prof. Mohammad Al-Suwiel, for their financial support.

I would like to thank Haesung Kim for implementing the graphical editor of DataLab.

Finally, very special thanks to my parents, to my wife Samia and to our sons Khaled and Abdulah for their unlimited support throughout my study in the USA. Their love, patient, and encouragement were the reasons for completing this thesis.

Table of Contents

List of Figures

List of Tables

DataLab: A Graphical System for Specifying and Synthesizing Abstract Data Types

Chapter 1

Introduction

This thesis examines an approach for the specification and synthesis of abstract data types (ADTs). It also describes the design and implementation of a model of this approach called DataLab.

Formal methods for specifying abstract data types (ADTs) (Gehani, 1982), such as algebraic specification, require great effort and a high level of expertise. In this research an approach is presented to specify ADTs using a combination of graphical and textual forms. In our approach, data structures and operations of ADTs are specified and created by direct manipulation, and operations are represented by a set of "Condition/Action" transformations.

The theory of program transformation is used in this system, but instead of transforming pieces of code from one language to another, e.g. from algebraic specification to horn-clauses, we present an entirely different approach -- transformation of visual representations into textual ones. This problem has been addressed most recently in ThinkPad (Rubin, 1985), (Rubin, 1986). We extend the method of ThinkPad by adding composition. The addition of composition allows one to design complex ADTs that go far

beyond ThinkPad's capability. In addition, DataLab uses a source code generator to produce an encapsulated imperative code (Pascal modules[1]) for each ADT[2] rather than a declarative code (PROLOG assertions).

Visual programming (Shu, 1988), (Chang, 1987) has not been shown to improve programmer productivity nor maintainability of code. Some of the limitations of visual systems can be seen in the work of Raeder (Raeder, 1984) and Glinert (Glinert, 1984). DataLab does not attempt to address the question of the usefulness of visual programming. Instead, it attempts to study the degree of usefulness of both text and graphics in the specification of ADTs. Therefore, DataLab uses text where most appropriate, and graphics where most appropriate.

1.1 The Problem

This thesis focuses on the problem of how to specify abstract data types through a combination of graphical and textual forms and how to automatically generate a compilable code from such specifications. The thesis claims that it is indeed possible to specify ADTs through such a combination of forms and to generate code from those specifications. It shows an approach to do this.

---

[1]DataLab depends on the syntax of a non-standard dialect of Pascal called LightSpeed Pascal (THINK, 1988). LightSpeed Pascal is heavily extended to support modules called units, which can be used to code an ADT.

[2]In this thesis, the two terms "ADT" and "Module" will be used to mean the same thing.

The proof of this conjecture is by demonstration. That is, a system was built that graphically and textually specifies ADTs and then automatically synthesizes the ADTs as Pascal modules.

The developed system is capable of doing the following:

1. Specify simple general data structures, e.g. arrays, linked lists, and trees;
2. Specify the operations that are to be performed on these data structures;
3. Specify complex ADTs by composition of simple ones, e.g. meshes and hash tables; and
4. Generate Pascal code from the specifications of the ADTs. The generated code encapsulates both the data structures and operations on them.

1.2 Others' Work

Programming languages have evolved over the last thirty-five years from first generation programming languages to fourth generation programming languages. One characteristic of most of these languages is that they use one-dimensional and textual formats to represent programs. These formats are suitable for sequential machines but not for the multi-dimensional and visual human mind (Chang, 1986). Visual programming is an attempt to simplify programming and to make better use of the human mind's capabilities, George Raeder states "... human mind is strongly visually oriented and that people acquire information at a significantly higher rate by discovering graphical relationships in complex pictures than by reading text. ... When we scan paragraph, headlines, and material in bold type to speed up text search, we are really acting in the

pictorial mode to overcome this limitation" (Raeder, 1985). As mentioned earlier, visual programming has not been shown to improve programmer productivity nor program maintainability, but we think a combination of text and graphics can make programming easier and more understandable.

Another attempt to simplify programming is the programming by example approach. This approach has attracted the attention of researchers in the programming community for its simplicity and naturalness. Nan Shu states "Most people are much better at dealing with specific, concrete objects than with abstract ideas. Examples and demonstrations are concrete and specific. When cleverly chosen and properly used, they help people understand the abstract concepts. Programming by example or demonstration attempts to extend these ideas to programming. From the user's point of view, the fundamental idea underlying programming by example or demonstration is really quite simple. In essence, the user writes a program (i.e., makes a specification) by giving examples to the computer of what he wants the program to do" (Shu, 1988).

A number of systems combine visual programming and programming by example to achieve the benefits of both approaches. In the remainder of this section we present some of these systems. An extensive review of literature is provided in chapter 2.

Pygmalion (Smith, 1975) is one of the first systems to combine visual programming and programming by example. In this system operators such as +, *, and <, and data such as integer numbers are represented by icons. It also has icon representation for conditional and loop statements. For example, to multiply two numbers, the user uses the mouse to create three icons, one for the operator and two for the operands. Then the user

moves the operand icons into the multiplication icon and performs the operation, which replaces the operator icon by the result icon. Programs are specified by examples, where each example shows the step of the computation on real data. The user's manipulations of the icons are recorded and stored in another icon. Pygmalion then plays back the recorded steps to perform the desired computation on the data. Pygmalion is used to specify small programs and not to specify ADTs.

"Programming by Abstract Demonstration" (Curry, 1978) is an extension of Pygmalion (Smith, 1975). As in Pygmalion, both data and operators are represented by icons, but the user cannot create new icons. The user can record programs by examples using either concrete data values such as an integer, an abstract data type such as a numerical subrange, or even a completely unspecified data value. The behavior of the program can be examined by executing it abstractly. For example, writing a program that adds a number to a numerical subrange produces a numerical subrange as the result. This system has the same limitations as Pygmalion.

Pygmalion (Smith, 1975) also provided the inspiration for yet another system, called "Programming by Example" (Halbert, 1984). To implement his system, Halbert uses a prototype version of the star system (Smith, 1982), called SmallStar, written by David Smith and Steve Hays, both of Xerox. SmallStar is basically an interactive graphical office information system, which includes icons for things like folders, printers, in-and-out boxes, and so on. Halbert's system extends SmallStar by adding three more icons, a trash-can icon, a calculator icon, and a program icon. The program icon is used to write programs by examples. Halbert's system provides a mechanism to record sequential user actions. To write a program, a user opens a program icon and selects the start recording

command in its window. He then defines the program by selecting objects with the mouse and pressing a command key that affects the selected objects. As the user does this the system records his or her actions and creates the program. When done, the user selects the stop recording command and may then run the program by selecting the run command. This system is limited by its domain in addition to having the limitations of the previously mentioned systems.

Laura Gould and William Finzer implemented a system called "Programming by Rehearsal" (Finzer, 1984). This system was created for curriculum designers to help them design and implement educational software for students and to program other educational activities. The system is based on the theater metaphor called the "rehearsal world". There are eighteen predefined performers grouped into five troupes. For each of these performers there is a set of predefined actions (cues). Creating a program (production) by rehearsal is done by selecting with a mouse some performers and copying them onto a stage. The designer then selects actions (cues) for the performers. When a designer has finished copying, placing, and directing the performers by selecting their actions, the program can be stored for later refinements or demonstration by a user. This system is limited by its predefined performers and operations and therefore is domain specific. It is used to write simple education-based programs and is not suitable for writing general-purpose programs.

Alan Borning implemented a system called ThingLab (Borning, 1981), which is a constraint-based programming by example system built as a graphical simulator laboratory. In ThingLab constraints are used to specify the relations that must hold among the parts of the simulation. ThingLab is not suitable for writing general-purpose programs.

Henry Lieberman described a system called Tinker (Lieberman, 1984), which is an experimental interactive programming environment for LISP. It uses examples both to write and debug programs at the same time. A program is constructed by demonstrating its steps of computation on real data, and Tinker displays graphically the results of each step. Tinker is used for writing and debugging programs and not for specifying ADTs.

"Programming in Pictures" (Raeder, 1984) is a system that combines graphical editing and programming by example to construct programs. It consists of four integrated editors, namely a picture editor which allows the user to draw data types, a type editor which allows the user to create data templates and to fill them with its data components, a function editor at the object level which allows the user to specify input and output examples for simple functions, and finally a function editor at the function level which can be used to create compound functions by combining simple ones and including conditional and loop constructs. The state of each editor is preserved between entries, and a global scratch pad is used to transfer entities between editors. This system is used to specify fairly simple programs and not to specify ADTs.

ThinkPad (Rubin, 1985), (Rubin, 1986) is a graphical system designed for programming by demonstration. Writing a program in ThinkPad consists of specifying a graphical representation for the program's data structures using the data editor, and demonstrating the operations on these data structures using the operation editor. Data structures are defined by selecting a graphical object from a list of predefined objects. Operations are demonstrated by pointing to and selecting a command from a menu or by pressing one of the buttons provided by the operation editor. It generates a PROLOG code by a one-to-one mapping between operators in ThinkPad and predicates in PROLOG.

ThinkPad is one of the main inspirations for DataLab. DataLab adopted ThinkPad notions of transformation to describe the behavior of functions. DataLab has extended ThinkPad's programming mechanism in a number of ways, most notably by allowing the specification of complex ADTs, by generating an encapsulated imperative code, by allowing data structures to be created and displayed in their classical notations, and by providing procedural control mechanisms.

1.3 The Solution

This research is oriented more toward creative design than toward mathematical or formal proof. Software tool designers do not yet know how to build tools for ADT fabrication because of several unsolved problems. Glinert (Glinert, 1986), (Glinert, 1987) refers to these open problems:

1. ...can suitable and uniform graphical-textual representations of data structures be found?
2. ...can [we] scale up ... to more than toy programs typically written by novice programmers?

The first problem is one of representation, while the second problem is one of size. These two problems plus the problem of how to generate code from representations are addressed by designing and building a system called DataLab.

DataLab is a testbed for exploring ideas on how to solve these three problems. It has two major components: a graphical editor and a source code synthesizer (see Figure 1.1).

The graphical editor allows the user to design and build data structures and operations on them by direct manipulation of graphical and textual representations of ADTs. The operations are specified as a set of "Condition/Action" transformations, which are also built by direct manipulation. The source code synthesizer takes the output of the graphical editor and generates a Pascal module for each specified ADT. The generated modules encapsulate both the data structures and the operations of the ADTs.

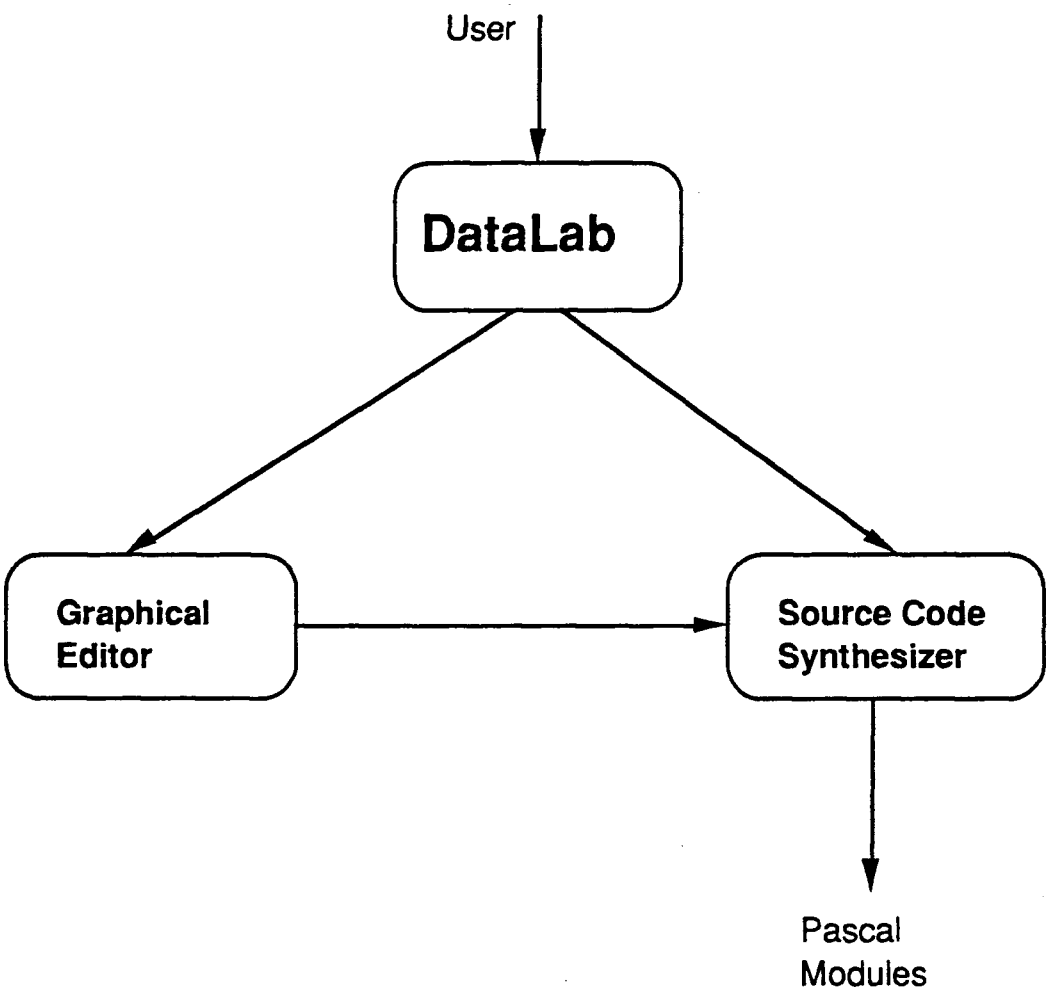


Figure 1.1: An overview of DataLab

Specifically the problems that must be solved to go beyond the current state of the art in program synthesis are listed below.

1. Representation. How should both static data structures and their dynamic operations be represented? We developed a set of icons to represent both static and dynamic data structures, and "Condition/Action" transformations to define the behavior of the operations.
2. Size. How do we synthesize complex ADTs? We developed a composition technique to permit construction of complex and heterogeneous data structures and their operations. Composition allows the specification of complex ADTs from simple ones.
3. Code. How is the intention of a programmer captured and used to generate code? We developed a source code synthesizer that generates Pascal modules from the visual/textual specifications of the ADTs.

DataLab incorporates innovative techniques for solving these three problems.

1.4 Contributions of DataLab

What makes this research significant? This question can be answered from several perspectives.

1. ADT perspective: Formal methods for specifying ADTs, such as algebraic specifications, require great effort and a high level of expertise. Our approach is simple enough to be used by any programmer. In fact, it is the

first system to produce an encapsulated imperative implementation of an ADT by direct manipulation.

2. Program Transformation perspective: The theory of program transformation is used in this research, but in a new way. Instead of transforming specifications from one language to another, we use an entirely different approach -- transformation of visual representations into textual ones. This problem has been addressed most recently in ThinkPad. We extend the methods of ThinkPad by adding composition. The addition of composition allows one to design complex ADTs that go far beyond ThinkPad's capability. In addition, DataLab produces procedural output rather than declarative output for each ADT.
3. Visual Programming perspective: Visual programming has not been shown to improve programmer productivity nor maintainability of code. This research does not attempt to address the question of the usefulness of visual programming. Instead, the research attempts to study the degree of usefulness of both text and graphics in the specification process. Therefore, DataLab uses text where most appropriate, and graphics where most appropriate. This research is not intended to solve all programming problems by substituting graphics for text.
4. Productivity perspective: This research makes a fundamental conjecture that programming can be made easier by using tools such as DataLab. More specifically, we claim that ADTs can be fabricated more quickly and reliably through DataLab than by hand. Future research will have to be conducted to support this conjecture under a controlled environment.

1.5 An Overview of the Dissertation

This dissertation is organized in the following manner. Chapter 2 presents a full survey of software development approaches. This survey covers most approaches ranging from those employing high-level languages to those which make use of direct manipulation. Chapter 3 describes the programming aspects of DataLab. It presents a formal description of the syntax, the semantics, and the computational model of DataLab. Chapter 3 also presents some examples to demonstrate the capabilities of DataLab. Chapter 4 presents the transformation method of DataLab. The transformation method includes the following: specification data structure, semantic data structure, semantic generator, and code generator. Chapter 5 presents the conclusion and recommendations for future research.

Chapter 2

Review of Literature

2.1 Introduction

Software development history goes back to the early days of computers, 35 years ago. Many approaches have been developed to implement software over this period. The following is a survey of software development approaches in the literature.

2.2 Third Generation Languages (3GLs)

This approach uses high-level programming languages to write computer software and is the most commonly used approach. Programming languages have evolved over the last 35 years, beginning with first generation languages, which are the machine languages, through second generation languages, which are the assembly languages, to third generation languages, which are the high-level languages such as FORTRAN, C, and Pascal, among others.

Using high-level languages to write a program has the advantage of producing an efficient object code. Code efficiency has been one of the major concerns of programming language developers over the years, beginning with the development of FORTRAN in the mid-1950s. As a result of this concentration on efficiency, many of the programming languages in use today produce a very efficient code.

However, high-level languages also have disadvantages. They are difficult to learn and use. For example, debugging a program written in one of these languages can be time consuming, especially in large and complex programs.

2.3 Application Development Systems

This approach uses a set of programming tools to create, design, and maintain applications. Examples of such systems are FORMAL and DF.

FORMAL (Shu, 1986) is a form-oriented and visual directed application development system for non-programmers. This system provides data manipulation capabilities which allow nonexpert users to program a wide range of data processing applications. FORMAL uses a form-oriented approach, where a form is used as a visual representation for data structures, data instances, and programs. A data processing activity can be expressed as a form process which takes one or two forms as input and produces a "form" as output. A program consists of form definitions and form processes.

DF (Ishida, 1986) is an application development system that uses the data flow concept. This system consists of a DF chart and a DF language that are used to design and develop business applications. The DF chart uses the data flow technique for describing the specifications of the application. The DF language generates COBOL or PL/1 programs from the DF chart specifications.

Most of the systems that use application development systems are domain-specific tools that are linked to data base management systems (DBMS).

2.4 Fourth Generation Languages (4GLs)

A program in a Fourth Generation Languages consists of a set of sequential statements and several mechanisms, such as filling in forms or panels, screen interactions, and computer aided graphics (Martin, 1986). These languages are mostly associated with data bases and are designed for a special class of applications, such as query languages, graphics packages, report generation, or creating data base applications. Examples of such languages are FOCUS, MIMER, NOMAD2.

FOCUS (Martin, 1986) is a fourth generation language for IBM machines, including the PC. It consists of a system of integrated tools that include graphics, screen generation, data query, report generation, functional modeling, and general application development tools. It supports various user interfaces, including English language, menus, and a command language with procedural statements.

MIMER (Martin, 1986) is a system of integrated tools that link into a relational data base management system. These tools include a relational DBMS, an end-user query language, an application program generator, a screen handler, an information retrieval system, a statistical package, and a graphics display package. MIMER generates COBOL or FORTRAN code which can be linked to user subroutines and compiled.

NOMAD2 (Martin, 1986) is a system of integrated tools that includes screen generation, report generation, graphics, statistical analysis, financial modeling, data base facilities, and procedural and nonprocedural languages for application development.

Examples of nonprocedural statements are the commands LIST and PLOT that are used to access and process data and to generate reports and graphs.

One disadvantage of the Fourth Generation language approach for application development is that these languages are designed for a specific class of applications, namely data base management systems. Another disadvantage is that the codes generated are inefficient and run slowly.

2.5 Very High-Level Languages (VHLLs)

This is a class of programming languages in which data structures are more abstract and complex than those available in languages like Pascal and FORTRAN. The use of these abstract data structures and their powerful operations improves programmer productivity and enhances program readability. The disadvantages of using VHLLs is the loss of efficiency in terms of program running speed. The VHLLs include languages like SETL, LISP and APL.

SETL (Schwartz, 1986) is a language based on the mathematical theory of sets. It is an acronym for SET Language. SETL uses some fundamental mathematical constructs, such as finite sets and maps. In SETL, a set is a value and its elements can be any values, including another set. The elements of a set do not have to be of the same type, and they can be nested to arbitrary levels. A set can be specified by listing the elements between two curly braces, or constructed using an expression called a "set former". SETL provides the standard set operations and the quantification operations (existential and universal). It also provides a map construct, which is a set of ordered pairs, and some of the constructs

found in Pascal-like languages, such as "if", "while", and "for" statements. For example, a program that finds the median of a set "s" can be written in SETL as follows:

```
program find-median;
    read(s);
    if exist x in s | #{y in s | y < x} = #{y in s | y > x}
    then
            print('The median is :' , x);
    else
            print('No median, the set s has an even number of elements');
    end;
end program find-median;
```

APL (Polivka, 1975), (Budd, 1988) is a VHL language that uses the "array" abstract data structure. One form of the array in APL is the vector, which is an ordered set of data. The elements of the vector can be numbers or characters and can be operated on by any of the APL primitive functions. In addition to arithmetic, relational, and assignment operators, APL has some special functions, such as catenation and index generator, that can be used to create and manipulate vectors. Another form of the array is the matrix, which is a two-dimensional array. A higher-dimensional array is also provided by APL. Arrays can be operated on by arithmetic, relational, and assignment operators; and they can be specified by listing their elements or they can be created by using a function called "reshape". Indexing can be done to select a single element or a group of elements of an array. APL has many interesting functions that make it easy to work with arrays. In this survey they cannot be listed because it would take pages to do so. Interested readers can consult one of the books on APL (Polivka, 1975), (Budd, 1988).

LISP (Brooks, 1985) is a VHL language that uses the "list" abstract data structure. A list consists of a sequence of elements between a pair of parentheses. An element can be an atom or a list, and a list can be nested to an arbitrary level. The operation can be one of the predefined operations provided by the language, or it can be defined by the user. The computation process consists of applying the operation to the arguments. LISP is an acronym for list processing and is mainly used for symbolic processing in the field of artificial intelligence.

2.6 Program Synthesis

Program synthesis is the construction of a program from a set of formal specifications. Two different formal specifications for abstract data type have been proposed in the literature, operational and algebraic. Informal specification can be ambiguous and incomplete, whereas formal specification can be made precise and can be used to prove the correctness of the program implementation and the equivalence of different implementations (Gehani, 1982). However, developing a formal specification and determining if it is consistent and complete can be very expensive and time consuming, which is a disadvantage of this approach.

Jalote developed a system for generating an implementation of an abstract data type in C language from its algebraic specifications (Jalote, 1987). The algebraic specifications used for the system have two parts, syntactic and semantic specifications. The syntactic specifications provide information such as variable names, variable types, operation domain, and operation range. Semantic specifications define the meaning of the operations using a set of axioms.

A group of researchers in this area have taken an artificial intelligence approach to program synthesis. They use a formal specification together with a programming knowledge base to construct programs. The following program synthesizers are domain specific and don't address ADTs.

Monlar, Navrat and Vojtek developed a system that automatically generates the operations of an abstract data type (procedures and functions) (Monlar, 1987). The knowledge is expressed in the form of transformation rules. Their system allows two forms of specifications for the ADT operations. One specification is in the form of input/output examples. For example, a stack push operation is specified as follows:

```
DECLARE PUSH(ATOM STACK) = STACK
OPERATION (PUSH(A ()) = (A))
        (PUSH(A (S D F)) = (A S D F))
ENDOPERATION
```

This has to be done for every operation. The other specification form is an input/output condition pair. For example, a program to determine equivalence between two sets is specified as follows:

```
COMPUTE
        (AND (ISIN S1 (ALL S2)) (ISIN S2 (ALL S1)))
WHERE
        ((SET S1) (SET S2))
```

The knowledge is represented in the form of transformational rules and includes general programming techniques, such as how to decompose a goal into two subgoals, and a

specific knowledge about the problem being solved. The system has been implemented in LISP and generates LISP programs.

Dromey describes a method to develop a number of sorting algorithms from a single specification (Dromey, 1987). The method basically applies different transformational rules to a single sorting specification to derive a number of sorting algorithms. The specification has both a precondition and a postcondition. For example, a specification for sorting a fixed set A of N integers using an array a[1 .. N] is

$$Q : N \geq 1$$
$$R : (\forall k)\,(1 \leq k < N \quad \Rightarrow \quad a_k \leq a_{k+1}) \wedge \text{perm}(a,A)$$

A transformation rule that introduces a new free variable q is specified as

$$R : (\forall k)\,(1 \leq k < N \quad \Rightarrow \quad (q = k+1 \Rightarrow \quad a_k \leq a_q))$$

Wood implemented a system called "Interactive Program Synthesizer" (Wood, 1982). This system allows the user to enter program specifications interactively. The interactions between the user and the system are directed by a simple graphical interface. Each partially completed program description is displayed in a separate window. The system knowledge base includes general programming knowledge.

Pratsch and Steinbruggen reviewed and classified the rule-based transformation systems (Pratsch, 1983). They concluded that these systems are problem-oriented subsystems based on small sets of powerful rules allied with advanced metalanguages. For these systems to be manageable and powerful they must be specific for a certain group of

users and restricted to particular problem domains, such as sorting, parsing, and graph problems.

2.7 Others

This section introduces a set of approaches that eliminate some of the problems in the techniques previously discussed. Some of the approaches presented here use visual direct manipulation to describe the specification of the program.

Some other approaches described below also generate a high-level language code and therefore have the advantage of producing efficient code. A disadvantage of some of these approaches is that they are domain specific. Also, there is no standard way to write programs using these approaches.

2.7.1 Visual Programming

This approach is used by some systems to construct programs using visual objects. The approach introduces a level of abstraction higher than the program's concrete objects such as variables and constants. The visual objects are used here mostly to represent abstract constructs of the program, such as "while", "for", and "if" statements, or higher abstract constructs such as a "block" or a "program". These systems do not allow the user to directly manipulate the program's concrete objects. Also, it is not easy to prove the correctness of a program written in this approach.

Some other systems use this approach only to help in writing, describing, and debugging programs. We will see examples of both systems.

VIGRAM (Hsieh, 1988) is one of the components of the OSU system (Lewis, 1988). This is a graphical editor which allows the programmer to define abstract programs and data structures, view the abstract programs, and write a textual form upon command. The fundamental elements of VIGRAM are a single-entry box and single-exit box. To represent sequence, boxes are connected left to right. An if-then-else construct is represented by a conditional box which is connected to a then-box and to an else-box. The case construct is represented by a tree-like box. To represent a loop construct (while, for, and repeat), a loop condition box is used. This is vertically connected to an action box which includes all the actions to be performed in the loop.

OMEGA (Powell, 1983) is a graphical system developed by Powell and Linton. This system allows the programmer to construct programs by pointing at predefined graphical constructs and moving them around on the screen. The system uses multiple windows to construct different pieces of the program, which later can be put together in one of the windows. Some of the windows are used only to view the predefined graphical objects. The graphical objects are visual representations for code or data. The actual code and data have to be specified textually.

PICT (Glinert, 1984) is a graphical programming language developed by Glinert and Tanimoto. In this system a program is represented as flowchart, and both types and number of variables are restricted to four six-digit, nonnegative decimal integers. The user constructs programs by selecting an icon from a menu of icons. This menu includes icons for program and subprogram construction, parameter passing modes, data structures, variables and operations. The control structures, such as while and repeat constructs, are

represented by colored directed paths. The system has four fundamental icons: programming, erasing, icon editor, and user library. PICT is used to build small programs.

PEGASYS (Moriconi, 1985), (Moriconi, 1986), is a graphical programming system developed by Moriconi and Hare. The system uses graphics for system design and documentation but not for program construction. The system allows the user to describe how the program is put together by means of a hierarchically structured collection of pictures. These pictures are mapped into logical formulas, and the system checks wether this hierarchy of pictures is logically consistent or not. The system provides a visual aid to the program construction, and the program itself is written textually.

It was mentioned at the beginning of this section that some systems use the visual approach to help in writing, describing, and debugging programs. Examples of these systems are given next.

PV (Brown, 1985) is a graphical system that can be used during program execution to display graphics that represent data structures such as variables, arrays, linked lists, and trees. The user can see the values of the variables changing and can monitor the control sequence of his program. The system allows the user to create pictures and to link them to code and data. The system is used to illustrate program execution.

PECAN (Reiss, 1984) is a graphical system that uses multiple windows to show different aspects of the program and its execution states. In this system, the program is represented internally as an abstract syntax tree, and the user sees views of this syntax tree.

These views include a symbol table, a data type definition, an expression tree and control flow graphs. The system supports three different views of program execution, control, program, and data.

VIPS (Isoda, 1987) is a visual debugger for ADA programs. This system uses graphics to show the static and dynamic behavior of program execution, employing multiple windows to show several views of program execution. These views include data, program text, block structure, acceleration, figure definition, interaction, and editor.

BALSA (Brown, 1985) is a system that enables the user to see an algorithm in action. It is used to animate algorithms, such as insertion sort, random number generator, and sequential search.

INCENSE (Myers, 1983) is a graphical system that allows the user to display data structures. The graphical displays for the data structures can be specified by the user or a default display can be used. The default displays provided by the system include literals for basic types, actual names for scalar types, stacked boxes for records and arrays, and curved lines with arrow heads for pointers. This system is mainly used for program debugging purposes. Other systems that use this approach can be found in (Chang, 1987), (Shu, 1988), (Reiss, 1987), and (Raeder, 1985).

2.7.2 Direct Manipulation

In this approach the concrete objects of the program, such as variables and constants, are represented by graphical objects. These are directly manipulated to define operations on

the concrete objects. One advantage of the approach is that the user writes a program by immediately manipulating the program's concrete objects, which makes programming easier and more understandable. A disadvantage is that it is not easy to prove the correctness of the program implementation using this approach.

The following sections introduce some of the software development approaches that use direct manipulation to construct programs.

2.7.2.1 User Interface

This approach uses direct manipulation to develop user interfaces. Developing a domain-specific interface is a complex task and needs new techniques. The following systems are examples of new techniques developed for user interface generations (Hix, 1989).

OSU (Lewis, 1988), an acronym for Oregon Speedcode Universe, is a system for programming by direct manipulation rather than by traditional programming languages. This system was developed at Oregon state university and allows the user to automatically produce user interface interactions and generate code for the specified interface.

UIDE (Foley, 1989) is a system built around a knowledge-based representation of an interface's conceptual design. The name is an acronym for User Interface Development Environment. It uses an interface design language (IDL) to define the interface and to list the interface information provided by the designer. By using a transformation algorithm, UIDE can automatically generate an alternative design with the same functionality as the

original one. This enables the user to design many functionally equivalent interfaces for the same application.

PERIODT (Myers, 1986) is an interface designing system that allows the user to demonstrate how the interface should look and work by specifying a set of examples. The system generates code for the specified interface.

2.7.2.2 Data Structure

In this approach program data structures are represented by graphical objects. The operations on these data structures are described by directly manipulating the graphical objects, and the program is described by a set of graphical examples or input-output specifications. Some of the systems presented in this section are discussed in Chapter 1, and repeated here for completeness.

Pygmalion (Smith, 1975) is one of the first systems to combine visual programming and programming by example. Programs are specified by a set of examples that show the steps of computation on real data. It is limited to simple programs and addresses neither generation of executable code for the specified programs nor ADTs.

Curry implemented a system called "Programming by Abstract Demonstration" (Curry, 1978). This system extended Pygmalion by allowing examples to include abstract data, but otherwise it has the same limitations as Pygmalion.

Halbert used the SmallStar office information system to build a system called "Programming by Example" (Halbert, 1984). Halbert's system is limited by its domain in addition to having the limitations of Pygmalion.

"Programming by Rehearsal" (Finzer, 1984) is another interesting system that was created to help curriculum designers to design and implement educational software for the student and to program other educational activities. This system is also limited by its domain and by not addressing general-purpose programming and ADTs.

ThingLab (Borning, 1981) is a constraint-based programming by example system built as a graphical simulation laboratory. ThingLab did not address general-purpose programming or ADTs.

"Programming in Pictures" (Raeder, 1984) combines graphical editing and programming by example to construct programs. It is used to build fairly simple programs and does not address ADTs.

Tinker (Lieberman, 1984) is an experimental interactive programming environment for LISP. It uses examples to write and debug programs at the same time. Tinker also does not address ADTs.

ThinkPad (Rubin, 1985), (Rubin, 1986) is a graphical system for programming by demonstration. ThinkPad uses declarative semantics to map visual specifications of ADTs into PROLOG clauses. DataLab adopted ThinkPad notions of transformation to describe the behavior of functions. DataLab has extended ThinkPad's programming mechanism in a

number of ways, most notably by allowing the specification of complex ADTs, by generating an encapsulated imperative code, by allowing data structures to be created and displayed in their classical notations, and by providing procedural control mechanisms.

An artificial intelligence approach to programming by example has been taken by some other researchers in this area.

AutoProgrammer (Biermann, 1976) is a system which automatically generates programs from a sample calculation given by the user. The system has a simple graphical interface that can be used to define data structures and calculation examples. The system stores user interactions and then synthesizes a general program which is capable of executing the given example.

Biermann (Biermann, 1978) introduces a domain-specific algorithm for constructing a certain class of LISP programs from a set of examples of their input/output behavior. The input/output examples of the behavior are specified in terms of a LISP expression format. The class of LISP programs used in this algorithm is called "regular LISP", and Biermann gives a definition and a set of theories about this class.

Chapter 3

Programming Aspects of DataLab

3.1 Introduction

This chapter describes the syntax, semantics, and computational model of DataLab (Al-Mulhem, 1989.a) (Al-Mulhem, 1989.c). Also, it presents two examples to demonstrate some of the capabilities of DataLab.

In DataLab the ADT's interface part is defined textually, while its operations are defined graphically. To define an ADT's operation, the user selects an operation (procedure or function) and then defines the behavior of the operation as a set of Condition/Action transformations. These transformations are built by selecting graphical icons from a palette and placing them on the screen. The graphical specification for each operation is mapped into an internal representation from which the source code of the ADT is generated.

This chapter is organized as follows. Section 3.2 presents the computational model, section 3.3 describes the syntax and semantics of the icons, section 3.4 describes the syntax and semantics of the transformations, and section 3.5 describes the syntax of the operations. Section 3.6 describes composition in DataLab and presents an example of defining a complex ADT, and section 3.7 gives a summary of this chapter.

3.2 Computational Model

DataLab is based on the control-flow computational model (Kodosky, 1989) with a declarative (visual) example-based syntax. That is, the behavior of an operation of an ADT is specified by a collection of Condition/Action transformations. Each transformation consists of a sequence of control-flow steps which define a condition on a data structure and an action to be carried out if the condition is satisfied.

The Condition/Action transformations are based on program transformation rules, and so we call them transformations or cases. Within each transformation we show by example the conditions that need to be satisfied and the actions that need to be carried out if the conditions are satisfied. Each transformation is assigned a transformation number which corresponds to the flow of control within the operation. Also, each item within each transformation is assigned a sequence number which corresponds to the flow of control within the transformation. Operations are carried out in order by ascending transformation number, followed by ascending sequence number within each transformation.

3.3 Syntax and Semantics of Icons

One problem that we experienced in designing and implementing DataLab is the creation of meaningful and unambiguous graphical icons to specify both the data structures and the operations on ADTs. The motivation for choosing the icons that are shown in Appendix A was to emulate the classical notations for data structures that are commonly used by instructors in teaching computer science classes. For example, in a computer science class, a linked list is usually represented as a set of boxes connected by arrows,

where boxes represent the elements of the list and arrows represent the pointers that connect these elements. DataLab uses precisely this notation to represent an ADT's data structures as shown in Figure 3.1.

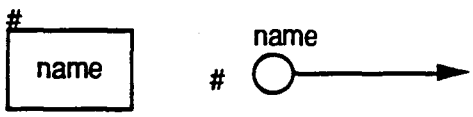


Figure 3.1: DataLab Icons: 1) A box for storage, and 2) an arrow for pointer.

The box is used to represent static storage structures, and the arrow is used to represent dynamic links. The number (#) associated with each icon represents the creation sequence of the object. These numbers show the order in which the icons are created; in general, the higher the number, the later the object is created.

There are four categories of icons in DataLab: objects, pointers, constants, and expressions/statements.

3.3.1 Objects

An object is displayed graphically as shown in Figure 3.2. Details of this icon are given in Appendix B.

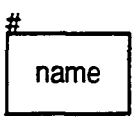


Figure 3.2: The object's icon.

An object icon can be used to represent parameters and variables. It can represent variables of simple types such as "integer", "boolean", "char", and variables of compound types such as "string", "subrange", "set", "array", and "record". The fields of a "record" and the elements of an "array" can be of type "pointer".

For example, a variable "temp" of type "integer" that is created as the sixth object in a transformation is represented as shown in Figure 3.3.

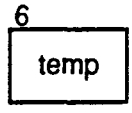


Figure 3.3: An icon for the variable "temp".

For variables of type "record" with fields of type "pointer" and variables of type "array" with base type "pointer", record fields and array elements are assigned using the "arrow" icon. For example, a record "Rec" with field "link" of type "pointer" that is assigned to "nil" is represented as shown in Figure 3.4.

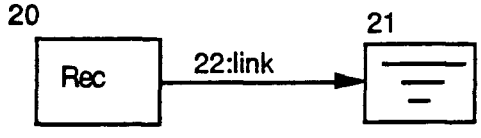


Figure 3.4: A record "Rec" with field "link" assigned to "nil".

3.3.2 Pointers

Pointers which are not assigned are represented graphically as shown in Figure 3.5. Details of this icon are given in Appendix B.

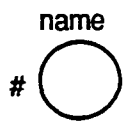


Figure 3.5: An icon for a non-assigned pointer.

For example, a pointer "list" of type "nodePtr", and sequence number "7," is represented as shown in Figure 3.6.

Figure 3.6: A pointer "list".

The "arrow" icon is also used to assign a value to a pointer. For example, "list" can be made to point to an object of compatible type. To make "list" point to an existing object "node" as shown in Figure 3.7, the user must first create "node" and then drag the pointer from "list" to "node".

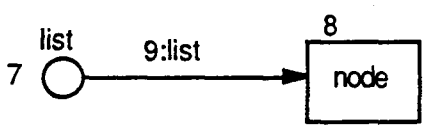


Figure 3.7: The pointer "list" points to an existing object "node".

The result of assigning "list" to a "nil" value is shown in Figure 3.8. First the "nil" object is created and then a pointer dragged from "list" to the "nil" box.

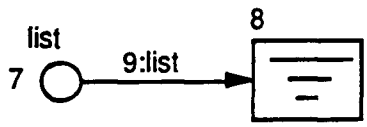


Figure 3.8: Assigning "list" to "nil".

Dereferencing "list" is done by dragging the pointer out of "list" to an empty location on the screen. DataLab will automatically create an object of compatible type and make "list" point to it, as shown in Figure 3.9.

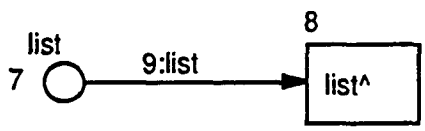


Figure 3.9: The pointer "list" is dereferenced.

A special object that is related to pointers is the "Don't Care" object, which is shown in Figure 3.10.

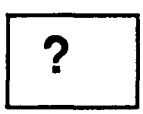


Figure 3.10: The "Don't Care" object.

"Don't Care" means that a pointer can be either "nil" or "non-nil". This is very useful in some transformations. For example, the condition shown in Figure 3.11, means that "tree" can be either "nil" or "non-nil".

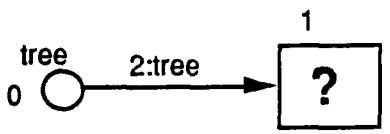


Figure 3.11: The "tree" pointer can be "nil" or "non-nil".

3.3.3 Constants

There are three predefined constants: 1) "nil", 2) boolean "true", and 3) boolean "false". They are shown in Figure 3.12.

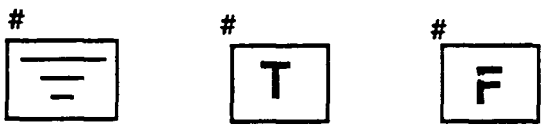


Figure 3.12: The constants: 1) nil, 2) true, and 3) false.

3.3.4 Expressions and Statements

This category is represented by the icon "ExplStmt". It allows the user to textually define any legal Pascal statement or expression including:

-- Assignments, e.g. x := 6;

-- Procedure calls, including recursive calls, e.g. compare(s1,s2);

-- Relational and logical expressions, e.g. tree <> nil; and

-- Input/Output statements, e.g. writeln('Current Node :', tree^.data).

"ExplStmt" represents an expression in the "Condition" part of a transformation, and a statement in the "Action" part. For example, the statement "writeln('Hi There');" is represented as shown in Figure 3.13.

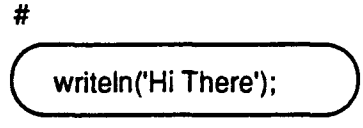


Figure 3.13: The "ExplStmt" icon.

3.4 Syntax and Semantics of Transformations

The behavior of an ADT's procedures and functions is described by a set of "Condition/Action" transformations. There are four types of transformations:

1) Conditional transformation.
2) Function transformation.
3) Loop transformation.
4) Sequence transformation.

A transformation consists of a set of conditions, a set of actions, and a set of control icons. The conditions and actions are build up from the icons described before (see section 3.3), and the control icons are used to separate the different parts of a transformation.

3.4.1 Control Icons

There are four control icons used in a transformation. The first one is the "start" icon which is used to begin any transformation, see Figure 3.14. The number(#) associated with the "start" icon is the transformation number.

Figure 3.14: The "start" icon.

The second control icon is the "transform" icon which is used to separate the "Condition" part from the "Action" part of a Conditional transformation. This symbol is shown in Figure 3.15.

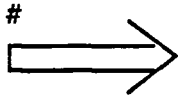


Figure 3.15: The "transform" icon.

The third control icon is the "return" icon which is used to separate the "Condition" part from the "Action" part of a Function transformation. This symbol is shown in Figure 3.16.

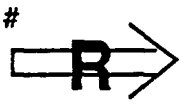


Figure 3.16: The "return" icon.

The fourth icon is the "loop" icon which is used to construct a loop transformation. This icon is shown in Figure 3.17.

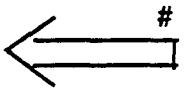
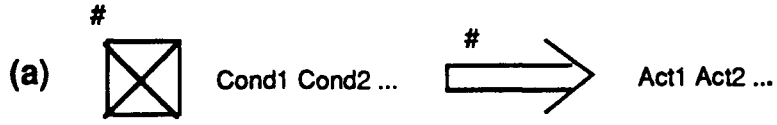


Figure 3.17: The "loop" icon.

3.4.2 Conditional Transformation

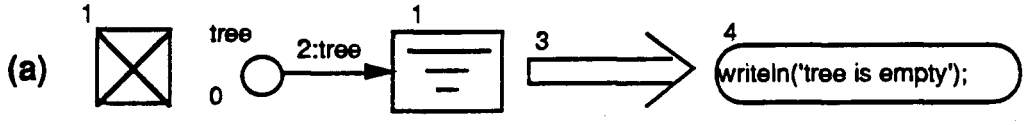
The general syntax of a Conditional transformation along with its meaning in Pascal is shown in Figure 3.18. "Condi" and "Acti" correspond to the i^{th} condition and the i^{th} action of the transformation, respectively.



(b) If (Cond1 and Cond2 ...) then begin Act1 Act2 ... end;

Figure 3.18: A Conditional transformation: a) syntax, and b) semantics.

Figure 3.19 shows an example of a Conditional transformation and its equivalent in Pascal.



(b) If (tree = nil) then begin writeln('tree is empty'); end;

Figure 3.19: An example of a Condition al transformation:a) syntax, and b) semantics.

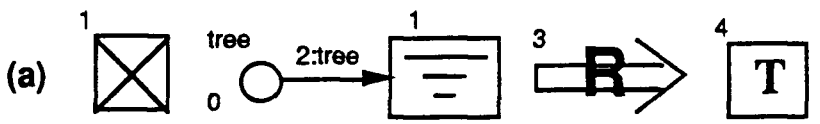
3.4.3 Function Transformation

The general syntax of a Function transformation along with its meaning in Pascal is shown in Figure 3.20. This transformation is used to return a single value from a function provided the conditions are satisfied.

(b) If (Cond1 and Cond2 ...) then *function-name* := Act;

Figure 3.20: A Function transformation: a) syntax, and b) semantics.

Figure 3.21 shows an example of a Function transformation and its equivalent in Pascal (the function name is "isEmpty").

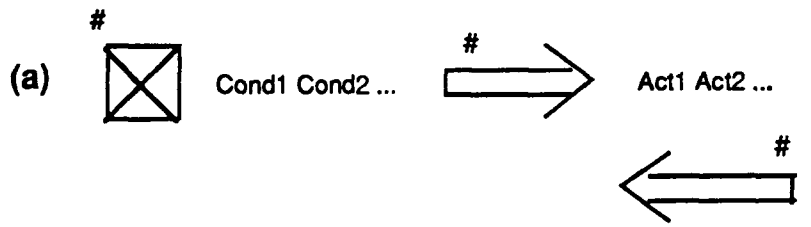


(b) If (tree = nil) then isEmpty := true;

Figure 3.21: An example of a Function transformation: a) syntax, and b) semantics.

3.4.4 Loop Transformation

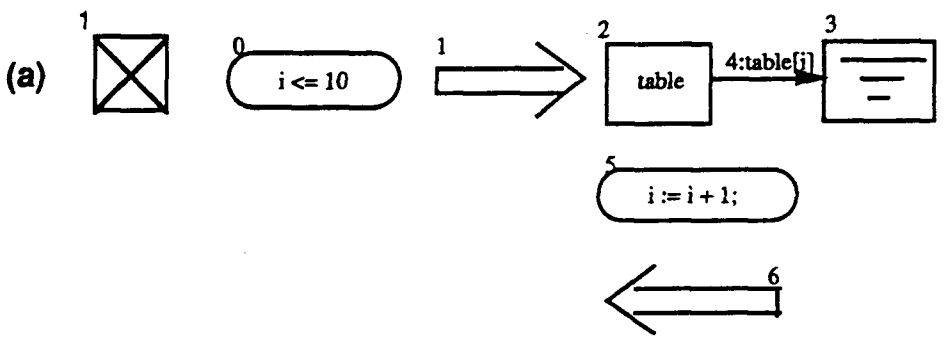
The general syntax of a Loop transformation along with its meaning in Pascal is shown in Figure 3.22. The last icon in this transformation must be the "loop" icon to indicate the end of the loop body.




```
(b) while (Cond1 and Cond2 ...) do
      begin Act1  Act2 ... end;
```

Figure 3.22: A Loop transformation: a) syntax, and b) semantics.

Figure 3.23 shows an example of a Loop transformation and its equivalent in Pascal.




```
(b) while (i <= 10) do
    begin
      table[i] := nil;
       i := i + 1;
    end;
```

Figure 3.23: An example of a Loop transformation: a) syntax, and b) semantics.

3.4.5 Sequence Transformation

The general syntax of a Sequence transformation along with its meaning in Pascal is shown in Figure 3.24. This transformation is used to represent a sequence of initialization statements in Pascal-like languages or a sequence of facts in Prolog-like languages.

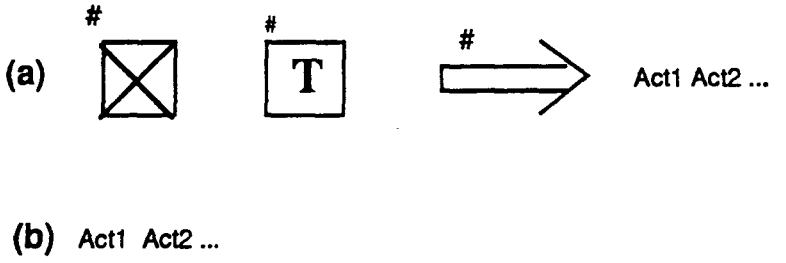


Figure 3.24: A Sequence transformation: a) syntax, and b) semantics.

Figure 3.25 shows an example of a Sequence transformation and its equivalent in Pascal.

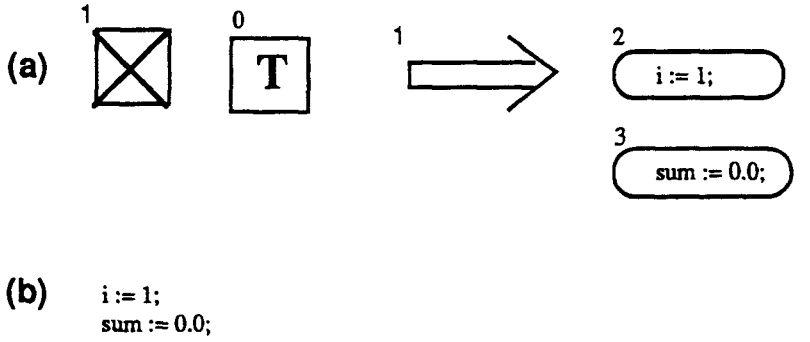


Figure 3.25: An example of a Sequence transformation: a) syntax, and b) semantics.

3.5 Operation Syntax

A BNF description for a single ADT operation in DataLab is shown in Figure 3.26.

```
operation -->   procedure
              | function

procedure --> proc_transformation_list

function --> func_transformation_list

proc_transformation_list -->   proc_transformation
                             | proc_transformation_list  proc_transformation

func_transformation_list -->   func_transformation
                             | func_transformation_list  func_transformation

proc_transformation -->   Conditional_transformation
                        | Loop_transformation
                        | Sequence_transformation

func_transformation -->   Conditional_transformation
                        | Function_transformation
                        | Loop_transformation
                        | Sequence_transformation

Conditional_transformation --> see section 3.4.2
Function_transformation --> see section 3.4.3
Loop_transformation --> see section 3.4.4
Sequence_transformation --> see section 3.4.5
```

Figure 3.26: An operation syntax.

3.6 Composition

One of the main contributions of DataLab is the use of composition to define complex ADTs. Composition is the process of specifying an abstract data type in terms of previously defined ADTs. In other words, it is the process of defining the data structures and operations of a certain ADT using definitions of data structures and operations of previously defined ADTs.

Systems such as ThinkPad (Rubin, 1985), (Rubin, 1986) are limited to defining simple ADTs such as trees, linked lists, and those often found in computer science books. Composition in DataLab allows one to specify and implement ADTs that go far beyond the capability of other systems.

Composition is a powerful programming technique which has been used by several traditional programming languages that support ADTs (Dahl, 1966), (Lampson, 1977), (Wirth, 1985). DataLab is the first graphical system that adopts composition in its graphical programming paradigm.

The next two examples demonstrate the use of composition in DataLab. First, we specify a linked list ADT, then we use the linked list ADT as a building block to specify a tree of linked lists ADT.

3.6.1 Example 1: A Linked List

The first step in defining a linked list is to enter the interface part as shown in Figure 3.27.

```
unit    LINKED_LIST;
interface
type
        LL_list   =   ^LL_node;
        LL_node   =   record
                      data   :   string;
                      next   :   LL_list;
                      end;
procedure LL_INSERT(var     list : LL_list; item : string);
implementation
end.
```

Figure 3.27: The interface specification of the linked list example.

The syntax of Pascal requires the type definitions of ADTs to be exported in order to accomplish data type instantiation. As a consequence, we specify the type definitions of the LINKED_LIST module as shown in Figure 3.27, and not in the implementation part (fully encapsulated). These type definitions are used in the TREE_NAMES module to instantiate different linked lists for each node in the tree. Appendix F shows several examples of fully encapsulated modules.

Next, the procedure "LL_INSERT" is specified by the two Conditional transformations in Figure 3.28 and 3.29.

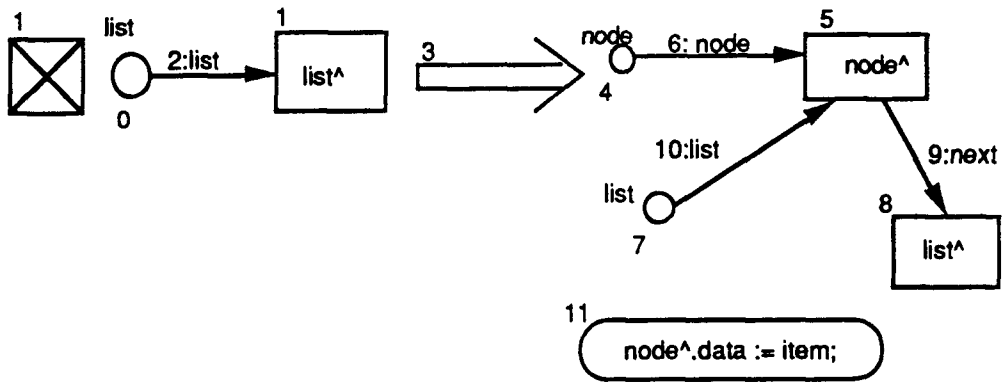


Figure 3.28: A transformation for a non-empty list.

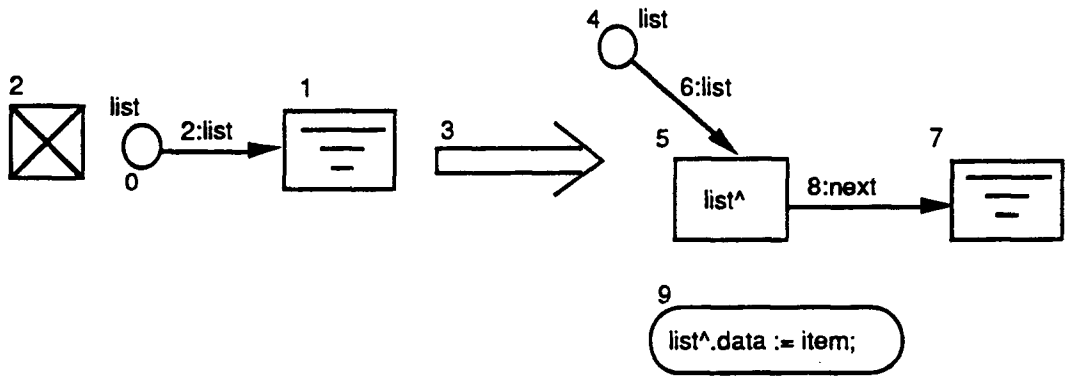


Figure 3.29: A transformation for an empty list.

The first transformation(Figure 3.28) handles the case when the list is not empty. It inserts a new node at the front of the list as follows. First, it creates "node" (sequence #4), then it assigns the field "next" (#9) to the first node in the list "list^" (#8), and reassign "list" to "node" (#10). Finally, it sets the field "data" to "item" (#11).

The second transformation(Figure 3.29) says that if "list" (#0) is empty (#1), then create a new list node and call it "list" (#4 dereferenced to #5), set its field "next" (#8) to

"nil" (#7), and set the value of the "data" field of "list" to the value of the parameter "item" (#9).

3.6.2 Example 2: A Tree of Linked Lists

Assume that we want to build the tree data structure shown in Figure 3.30 and to define some operations to manipulate it.

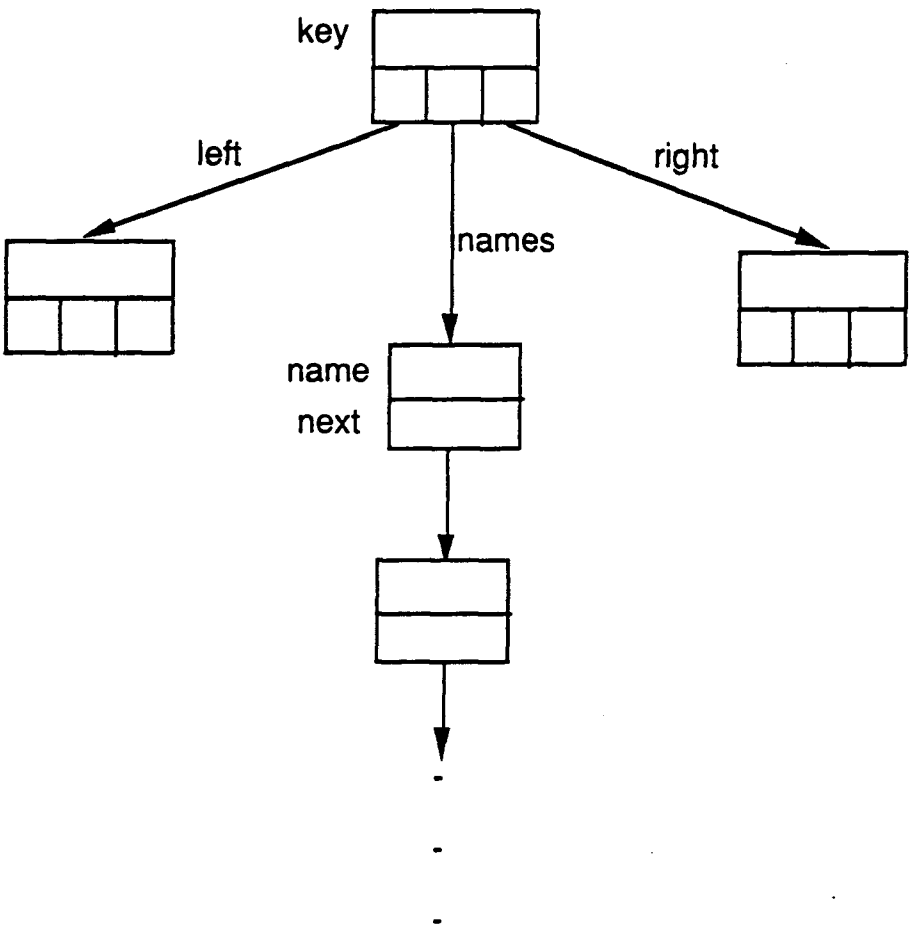


Figure 3.30: A tree data structure.

A node in the tree has four fields, "key," "left," "right" and "names." The "key" field stores an alphabetic character (a .. z), and "left" and "right" are the two pointers to the left and right subtrees, respectively. The "names" field is a linked list of names starting with an alphabetic character that matches the one stored in "key".

We show how to specify this ADT using the LINKED_LIST ADT defined earlier.

First, the interface part is entered as shown in Figure 3.31.

```
unit TREE_NAMES;
interface
uses LINKED_LIST;     {imported ADT}

procedure TN_INSERT(name : string);
procedure TN_INSERT_AUX(name : string);

implementation

type
        TN_tree = ^TN_node;
        TN_node = record
                    key : char;
                    left : TN_tree;
                    right : TN_tree;
                    names : LL_list;
                    end;

end.
```

Figure 3.31: The interface specification for the tree example.

The procedure "TN_INSERT" is specified by three Conditional transformations as shown in Figures 3.32-3.34.

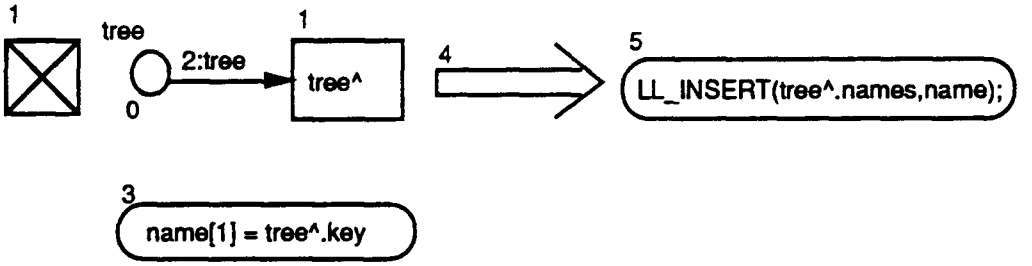


Figure 3.32: A transformation for a non-empty tree and the tree's key matches name[1].

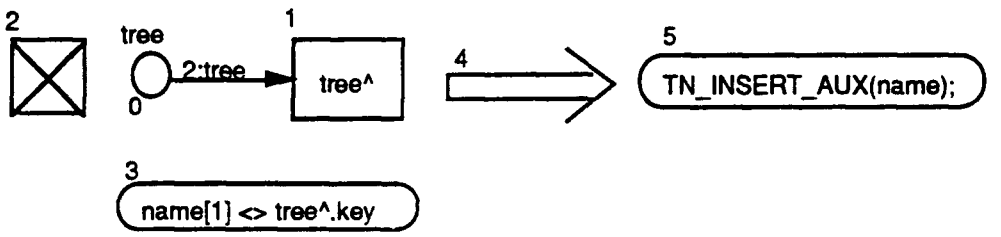


Figure 3.33: A transformation for a non-empty tree and name[1] doesn't match the tree's key.

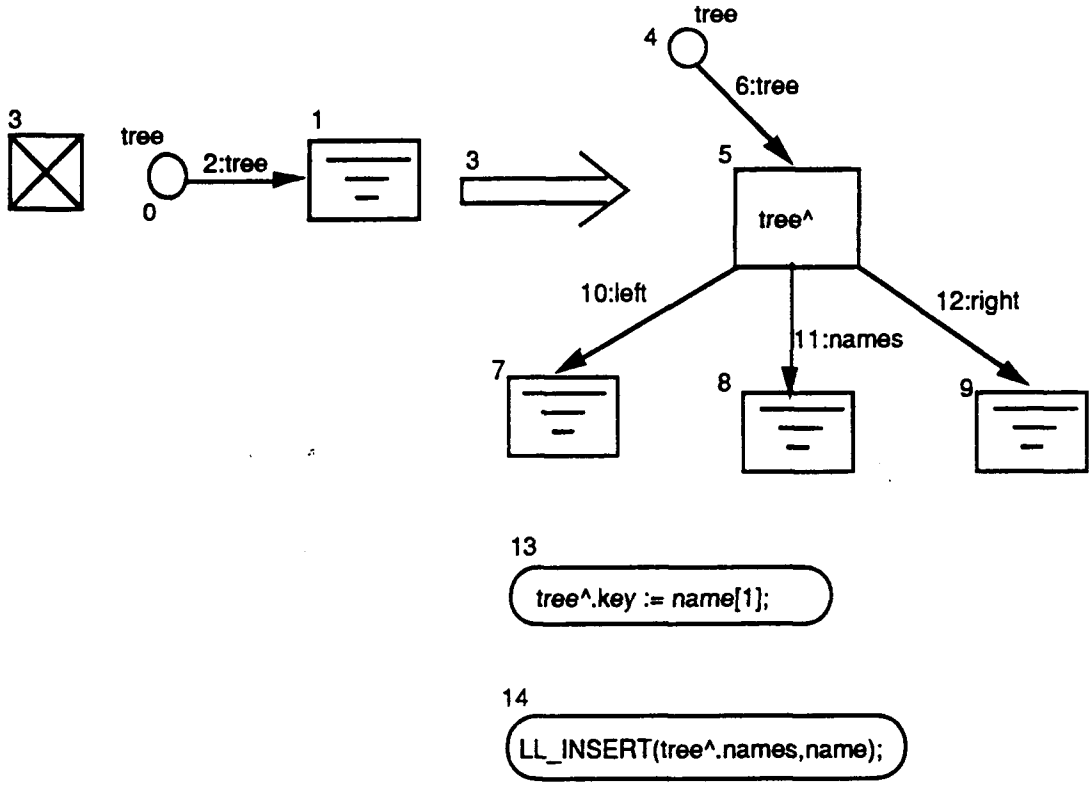


Figure 3.34: A transformation for an empty tree.

The first two transformations (Figures 3.32-3.33) deal with the condition where "tree" is not empty. There are two cases depending on the first character of the parameter "name":

1--The character equals the value in "key". In this case, "name" is inserted in the "names" list using the procedure LL_INSERT, which is defined in "LINKED_LIST".

2--The character doesn't equal the value in "key". In this case, "name" is inserted in its proper place by calling the procedure TN_INSERT_AUX.

The third transformation (Figure 3.34) specifies that if "tree" (#0) is empty, then do the following:

-- Create a new node, "tree" (#4), and initialize it by setting its "left" (#10), "right" (#12), and "names" (#11) fields to "nil".

-- Set the "key" field of the node "tree" to the first character in the parameter "name" (#13).

-- Call the procedure "LL_INSERT", which is defined in the LINKED_LIST ADT, with the "names" list as its first argument and the parameter "name" as its second argument (#14).

3.7 Summary

DataLab is a testbed for exploring the advantages of using text and graphics to specify abstract data types. It uses text and graphics where they are most appropriate, as follows. Text is used to specify an ADT's interface, because it contains static information without any actions such as type definitions. While it is simple to provide a graphical interface for specifying the ADT interface containing static information, this is a case where text appears to be quicker and simpler to use. On the other hand, graphics are used to specify data structures and ADT's operations because they contain actions such as pointer assignments and loops. The example-based approach, which has attracted the attention of researchers for years, is used for representing the ADT's operations.

The icons presented in this chapter are similar to the symbols commonly used in computer science classes; therefore, we think they are natural and easy to remember and use. We show how to use our icons in a combination with the example-based approach to specify complex ADTs, and we claim that such a combination is easy to use and should increase software productivity. We don't prove this claim; it is left as a problem for further research.

There are still many problems that need to be solved in order for a visual/textual approach to gain wide acceptance in the programming community. One problem is to develop a formal theory for such approaches. We think that visual/textual languages have advantages over textual languages, but they need tools to make them comparable in speed, memory usage, and flexibility to textual languages.

Chapter 4

Program Transformation Method of DataLab

4.1 Introduction

This chapter describes the program transformation method used in DataLab (Al-Mulhem, 1989.b). We present a model for transforming visual/textual specifications of ADTs to Pascal modules (THINK, 1988). This model includes two data structures: one for the ADT's visual/textual specifications and the other for the semantics of the specifications. It also includes two algorithms: one for mapping the visual/textual specifications into their semantics and the other for mapping the semantics into Pascal modules. A data flow diagram for the program transformation method in DataLab is shown in Figure 4.1.

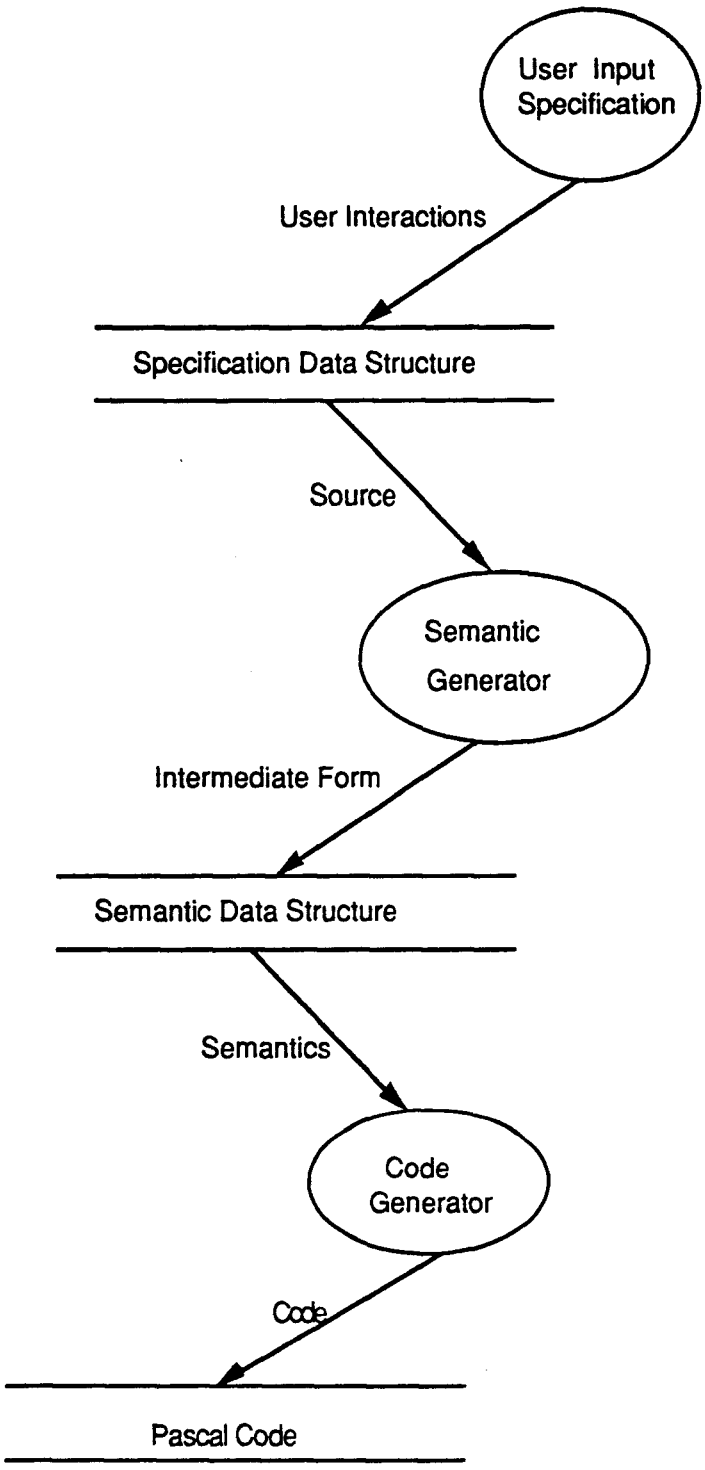


Figure 4.1: A data flow diagram for the program transformation method.

In Figure 4.1, the user's interactions with DataLab are stored in an internal data structure, called specification data structure because it contains a visual/textual specification of an ADT. For more information on the specification data structure and the graphical editor of DataLab see (Kim, 1989).

The semantic generator in Figure 4.1 maps the specification data structure into another data structure, called semantic data structure because it contains the meanings of the visual/textual specifications. Finally, the code generator maps the semantic data structure into a target source code.

This chapter is organized as follows. The next section describes the specification data structure, section 4.3 describes the semantic data structure, section 4.4 presents the semantic generator, and section 4.5 presents the code generator. Section 4.6 presents an example, and section 4.7 gives a summary of the chapter.

4.2 Specification Data Structure

The specification data structure is an internal representation for the visual/textual specification of an ADT. An overview of the specification data structure in DataLab is shown in Figure 4.2. A complete description of the specification data structure is documented in Appendix C.

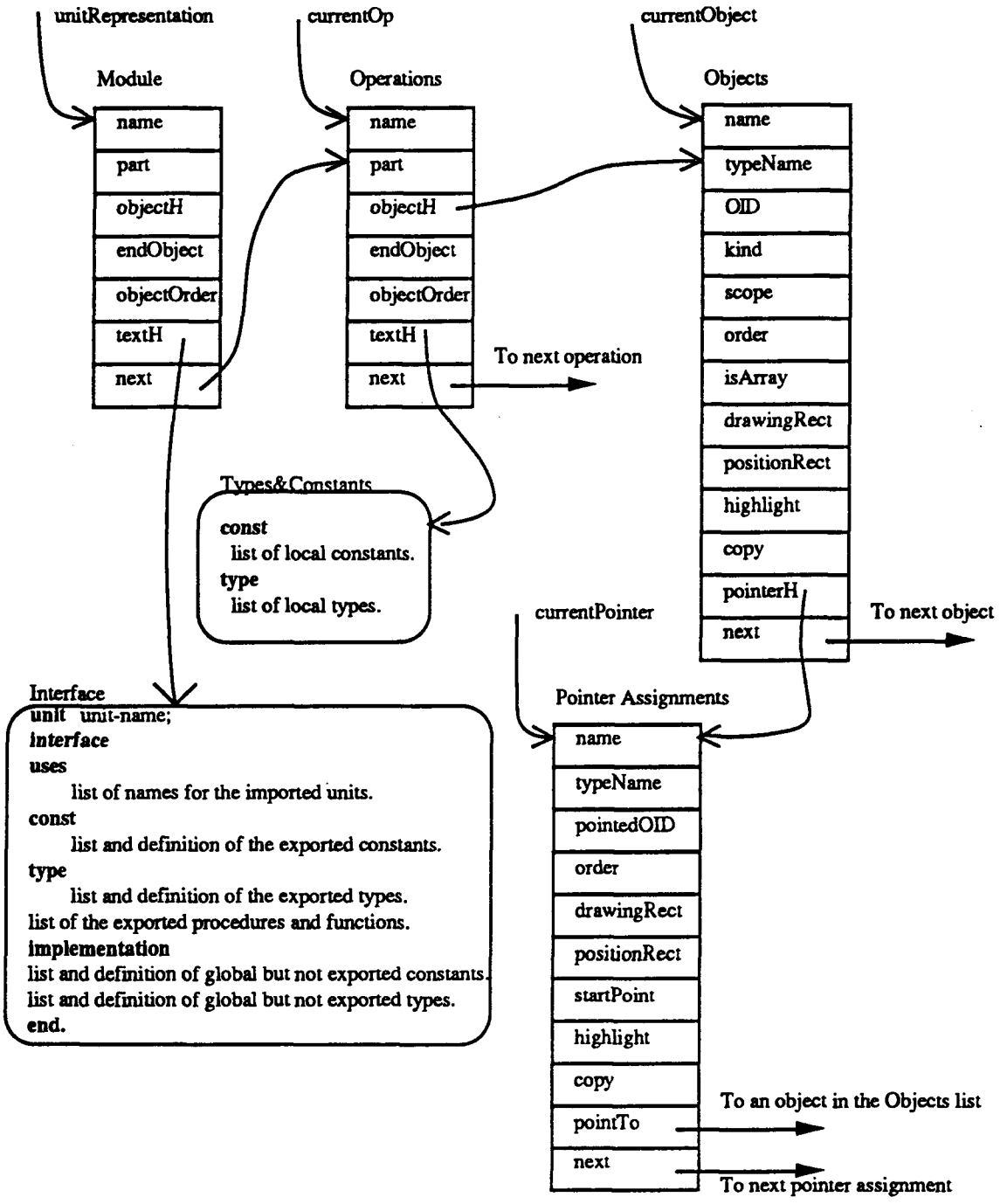


Figure 4.2: Specification data structure.

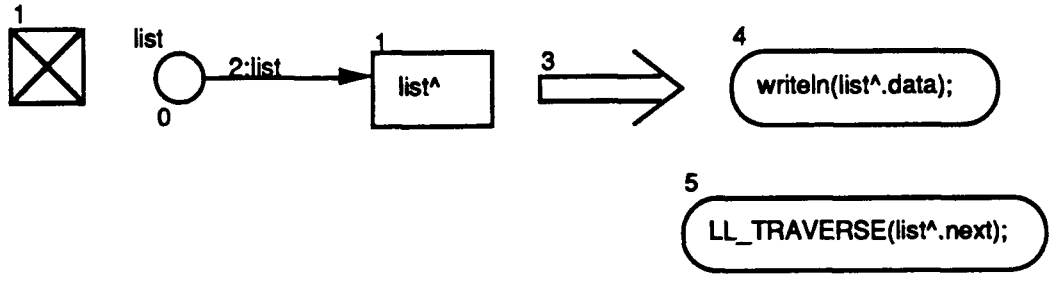
Note that in Figure 4.2, each operation has one "Objects" list which contain specifications for all the icons that define all the transformations of the operation, and the transformations are separated by their "start" icons. In other words, the "Objects" list of an operation begins with the "start" icon of one of the transformations, followed by the icons within that transformation, followed by the "start" icon of the next transformation and so on.

Also note that, the "Pointer Assignments" list is optional and not every icon has it. An icon of type "record" with fields of type "pointer" has a "Pointer Assignments" list to represent its field assignments. Similarly, an icon of type "array" with base type "pointer" has a "Pointer Assignments" list to represent its element assignments. Also, an icon of type "pointer" has a "Pointer Assignments" list with one node to represent the pointer assignment. Icons of other types don't have "Pointer Assignments" list (their "pointerH" field equals to "nil").

For example, suppose we want to specify the LINKED_LIST module (see chapter 3) with an operation called "LL_TRAVERSE" for traversing the contents of a linked list. Figure 4.3 shows the specification of the module, and Figure 4.4 shows the corresponding specification data structure.

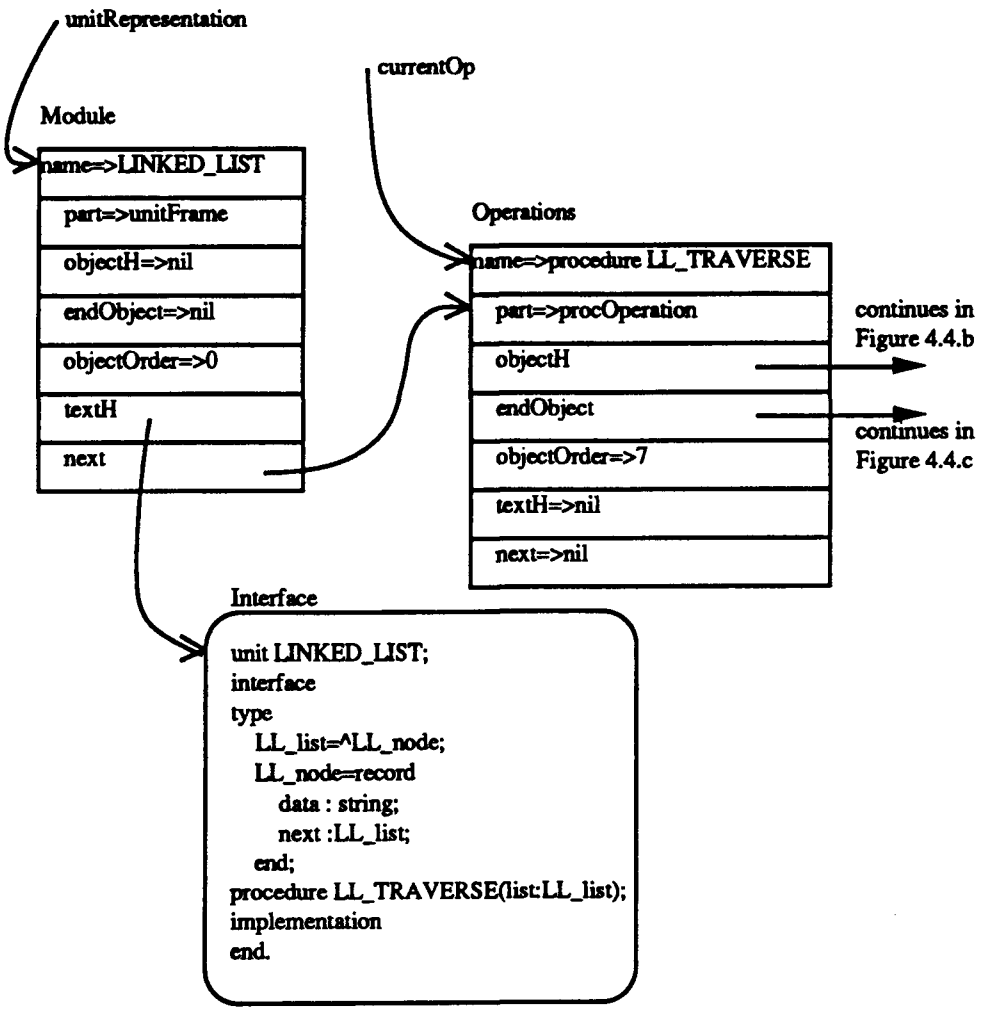
```
unit    LINKED_LIST;
interface
type
        LL_list   =   ^LL_node;
        LL_node   =   record
                        data   :   string;
                        next   :   LL_list;
                        end;
procedure LL_TRAVERSE(list : LL_list);
implementation
end.
```

(a) Interface specification.

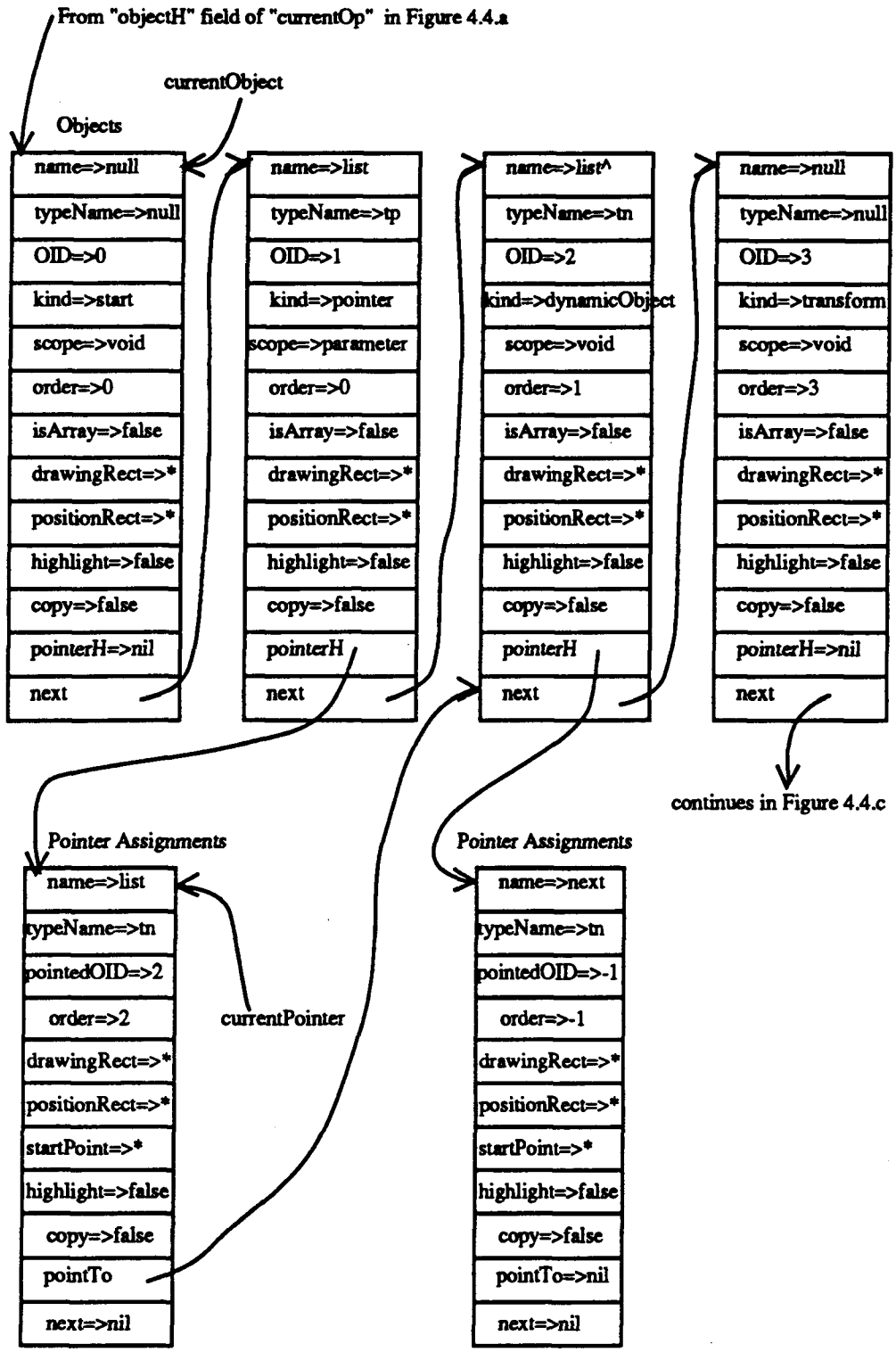


(b) Specification for the procedure LL_TRAVERSE.

Figure 4.3: The visual/textual specification for the LINKED_LIST module.

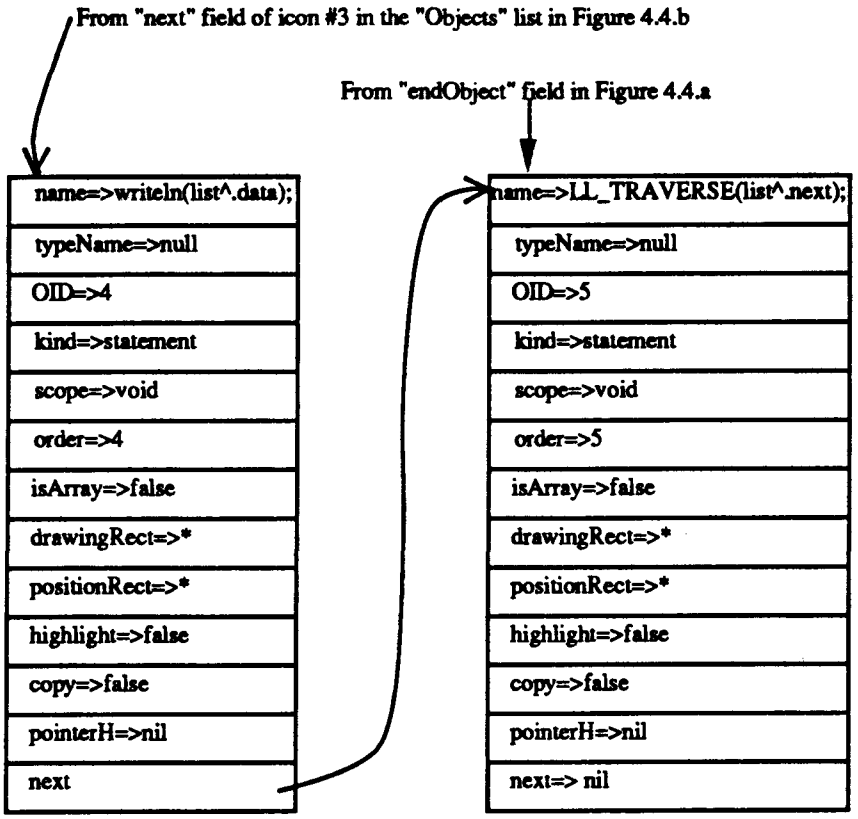


(a) Unit and operation representation.



* These fields contain the coordinates of the icon on the screen.

(b) Condition part of the transformation.



* These fields contain the coordinates of the icon on the screen.

(c) Action part of the transformation.

Figure 4.4: Specification data structure for the LINKED_LIST module.

4.3 Semantic Data Structure

A semantic data structure is an internal representation for the semantics of the visual/textual specifications of an ADT. An overview of the semantic data structure is shown in Figure 4.5. A complete description of the semantic data structure is documented in Appendix D.

Note that in Figure 4.5, the "Transformations" list contains the semantics of all transformations of an operation. Unlike Figure 4.2, where all transformations of an operation are stored in one linked list ("Objects" list), the semantics for each transformation is extracted from the "Objects" list of Figure 4.2 (see semantic generator) and stored in a separate element in the "Transformations" list of Figure 4.5. This is analogous to parsing an input sentence in a traditional language, and producing an intermediate form of the program.

Consider the binary tree module shown in Figure 4.3. The semantic data structure for this module is shown in Figure 4.6.

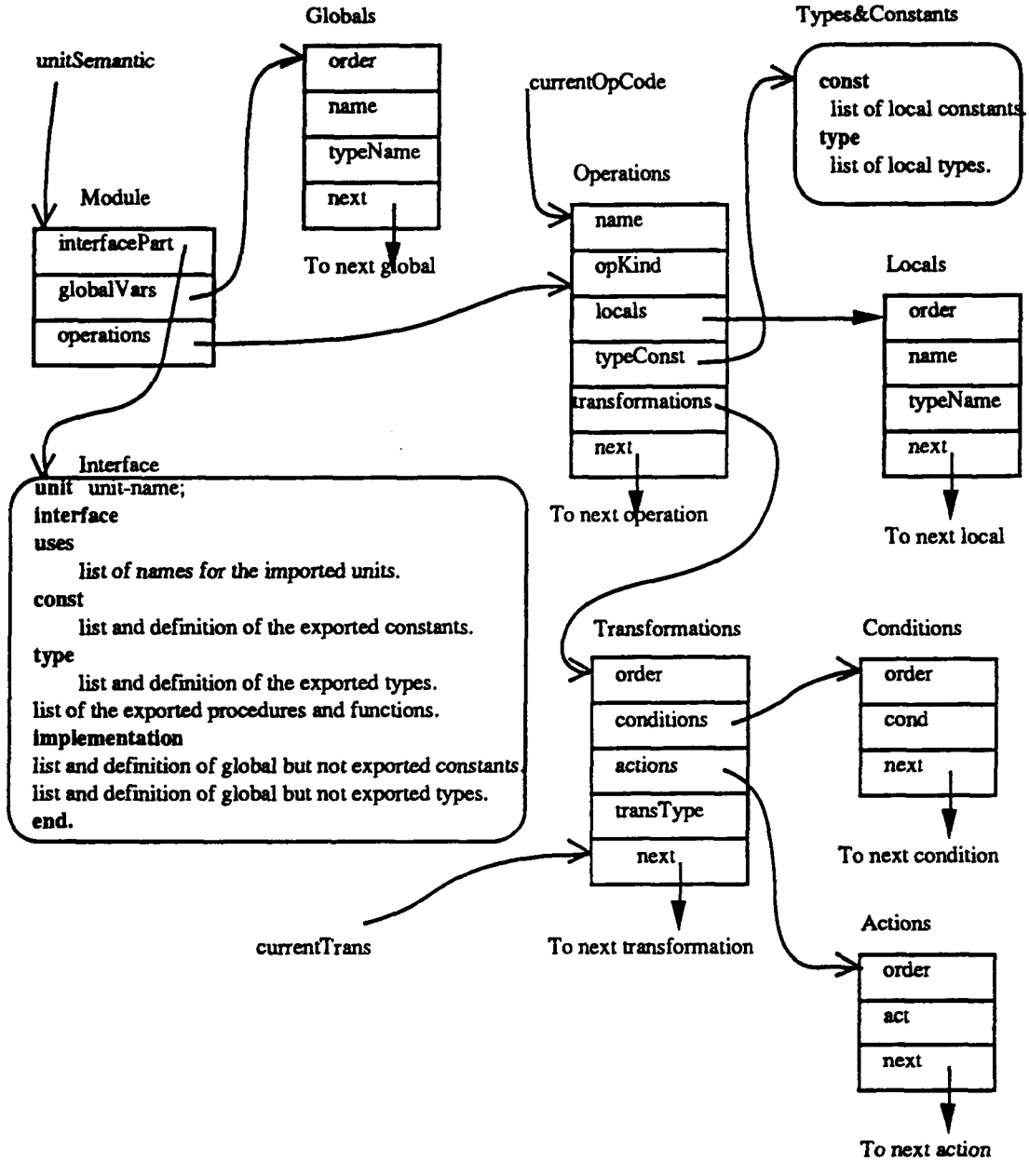


Figure 4.5: Semantic data structure.

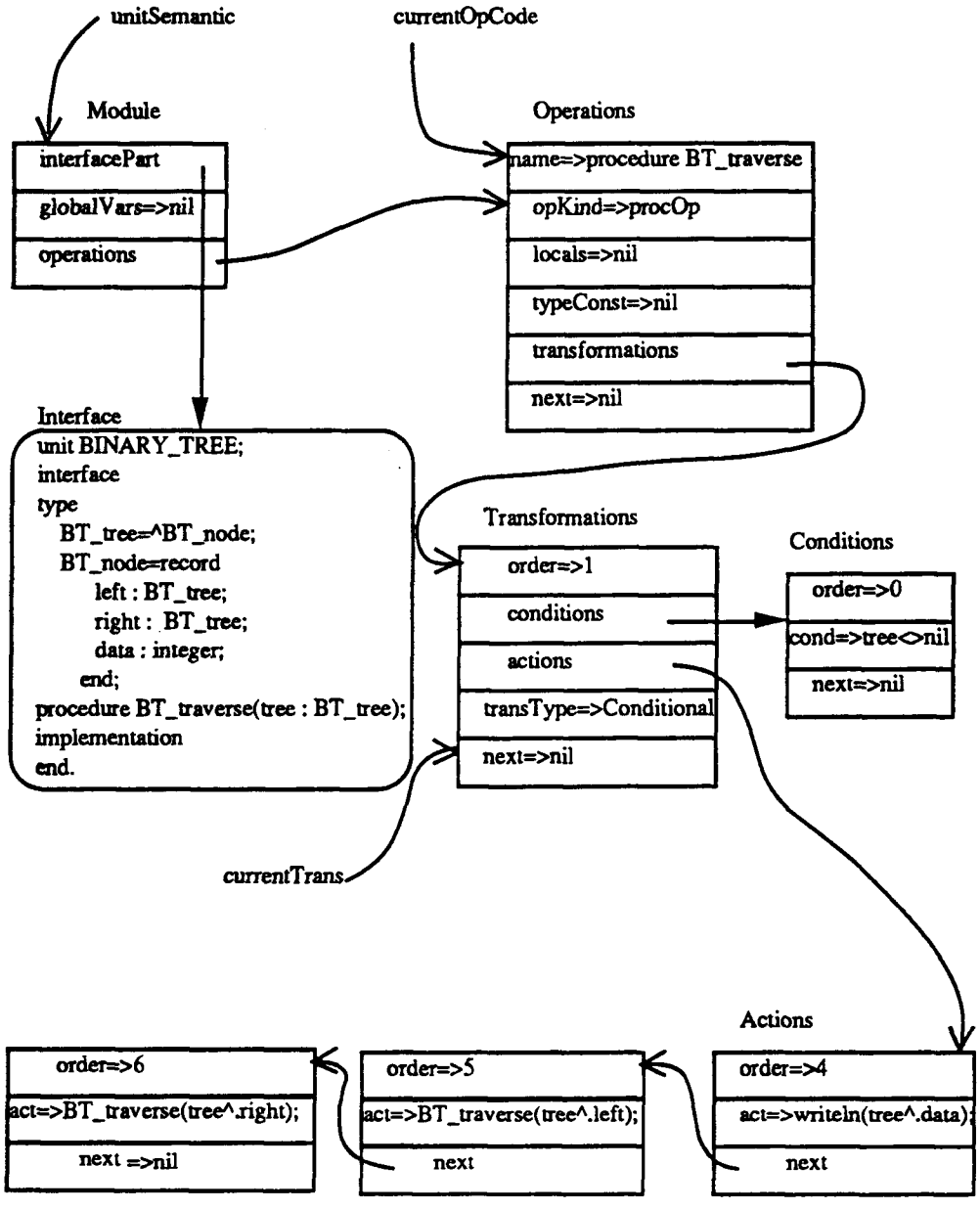


Figure 4.6: Semantic data structure for the LINKED_LIST module.

4.4 Semantic Generator

The input to the semantic generator is the specification data structure and the output is the semantic data structure as shown in Figure 4.1.

The semantic generator copies all Pascal syntax portions from the specification data structure to the semantic data structure without modification, because they are already in a form which can be compiled. Then it traverses the "Objects" list of the first operation in the "Operations" list of Figure 4.2 and runs the two-state Mealy machine of Table 4.1 to decide what actions to perform (Hopcroft, 1979). The semantic generator terminates when it reaches the end of the "Operations" list of Figure 4.2. A summary of the semantic generator is given in Algorithm 1.

Algorithm 1 - Semantic Generator.

Input: A specification data structure of Figure 4.2.

Output: A semantic data structure of Figure 4.5.

Method:

Copy the Pascal syntax portions of Figure 4.2 into Figure 4.5.

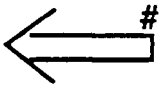
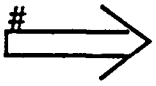
For all operations in the "Operations" list of Figure 4.2

For all icons in the "Objects" list of Figure 4.2

Run the Mealy machine of Table 4.1.

Terminate.

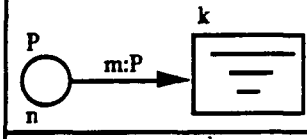
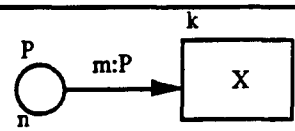
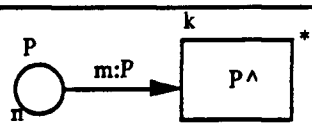
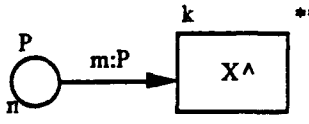
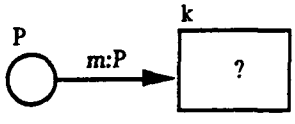
Table 4.1: Mealy machine for icons(continues on the next page).

State / Input	inCondition	inAction
# 	Begin a new transformation.	State := inCondition. Begin a new transformation.
	error	transType* := LoopTrans.
	State := inAction. transType* := FunctionTrans.	error
	State := inAction. transType* := ConditionalTrans	error
	Insert Exp** into the "Conditions" list.	Insert Stmt** into the "Actions" list.
	transType* := SequenceTrans.	Insert "true" into the "Actions" list.
	error	Insert "false" into the "Actions" list.

*transType holds the transformation's type, "LoopTrans", "FunctionTrans", "ConditionalTrans", and "SequenceTrans" for Loop, Function, Conditional, and Sequence transformations, respectively.

**Exp and Stmt are the expression and the statement entered by the user and contained in the "ExplStmt" icon respectively (see chapter3).

Table 4.1: Mealy machine for a pointer and a pointer assignment(continues from previous page).

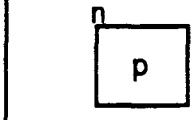
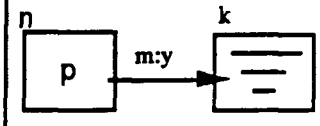
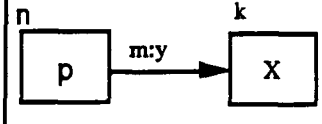
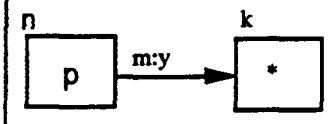
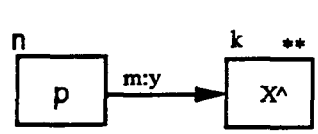
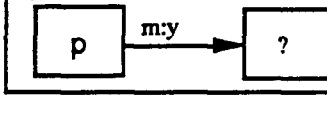
State / Input	inCondition	inAction
	Declare "p".	Declare "p".
	Declare "p" and Insert "p=nil" into the "Conditions" list.	Declare "p" and Insert "p:=nil;" into the "Actions" list.
	Declare "p" and Insert "p^=x" into the "Conditions" list.	Declare "p" and Insert "p^:=x;" into the "Actions" list.
	Declare "p" and Insert "p<>nil" into the "Conditions" list.	Declare "p" and Insert "new(p);" into the "Actions" list.
	Declare "p" and Insert "p=x" into the "Conditions" list.	Declare "p" and Insert "p:=x;" into the "Actions" list.
	Declare "p" and Insert "true"*** into the "Conditions" list.	error

* dereferenced icon for the same pointer "p".

** dereferenced icon for a different pointer "x".

*** optimization of the condition "p=nil and p<>nil".

Table 4.1: Mealy machine for a record with its field assignments, and an array with its element assignments(continouse from previouse page).

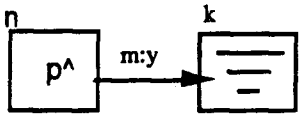
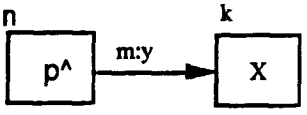
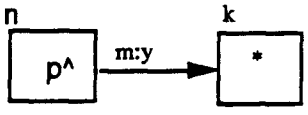
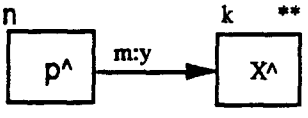
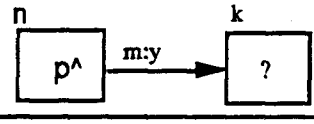
State / Input	inCondition	inAction
	Declare "p".	Declare "p".
	Declare "p" and if "p" is a record then insert "p.y=nil" into the "Conditions" list else if "p" is an array then insert "p[y]=nil" into the "Conditions" list.	Declare "p" and if "p" is a record then insert "p.y:=nil;" into the "Actions" list else if "p" is an array then insert "p[y]:=nil;" into the "Actions" list.
	Declare "p" and if "p" is a record then insert "p.y^=x" into the "Conditions" list else if "p" is an array then insert "p[y]^=x" into the "Conditions" list.	Declare "p" and if "p" is a record then insert "p.y^:=x;" into the "Actions" list else if "p" is an array then insert "p[y]^:=x;" into the "Actions" list.
	Declare "p" and if "p" is a record then insert "p.y<>nil" into the "Conditions" list else if "p" is an array then insert "p[y]<>nil" into the "Conditions" list.	Declare "p" and if "p" is a record then insert "new(p.y);" into the "Actions" list else if "p" is an array then insert "new(p[y]);" into the "Actions" list.
	Declare "p" and if "p" is a record then insert "p.y=x" into the "Conditions" list else if "p" is an array then insert "p[y]=x" into the "Conditions" list.	Declare "p" and if "p" is a record then insert "p.y:=x;" into the "Actions" list else if "p" is an array then insert "p[y]:=x;" into the "Actions" list.
	Declare "p" and Insert "true"*** into the "Conditions" list.	error

* dereferenced icon for the same record's field "p.y^", or the same array's element "p[y]^".

** dereferenced icon for a different pointer "x".

*** optimization of the condition "p.y=nil and p.y<>nil", or the condition "p[y]=nil and p[y]<>nil".

Table 4.1: Mealy machine for a dreferenced record with its field assignments, and a dereferenced array with its element assignments(continouse from previouse page).

Input \ State	inCondition	inAction
	if "p^" is a record then insert "p^.y=nil" into the "Conditions" list else if "p^" is an array then insert "p^[y]=nil" into the "Conditions" list.	if "p^" is a record then insert "p^.y:=nil;" into the "Actions" list else if "p^" is an array then insert "p^[y]:=nil;" into the "Actions" list.
	if "p^" is a record then insert "p^.y^=x" into the "Conditions" list else if "p^" is an array then insert "p^[y]^=x" into the "Conditions" list.	if "p^" is a record then insert "p^.y^:=x;" into the "Actions" list else if "p^" is an array then insert "p^[y]^:=x;" into the "Actions" list.
	if "p^" is a record then insert "p^.y<>nil" into the "Conditions" list else if "p^" is an array then insert "p^[y]<>nil" into the "Conditions" list.	if "p^" is a record then insert "new(p^.y);" into the "Actions" list else if "p^" is an array then insert "new(p^[y]);" into the "Actions" list.
	if "p^" is a record then insert "p^.y=x" into the "Conditions" list else if "p^" is an array then insert "p^[y]=x" into the "Conditions" list.	if "p^" is a record then insert "p^.y:=x;" into the "Actions" list else if "p^" is an array then insert "p^[y]:=x;" into the "Actions" list.
	Insert "true"*** into the "Conditions" list.	error

* dereferenced icon for the same record's field "p^.y^", or the same array's element "p^[y]^".

** dereferenced icon for a different pointer "x".

*** optimization of the condition "p^.y=nil and p^.y<>nil", or the condition "p^[y]=nil and p^[y]<>nil".

In Table 4.1, there are two states "inCondition" and "inAction". The input to Table 4.1 is either an icon in the "Objects" list as shown in Figure 4.7, or an icon in the "Objects" list with its pointer assignment in the "Pointer Assignments" list as shown in Figure 4.8.

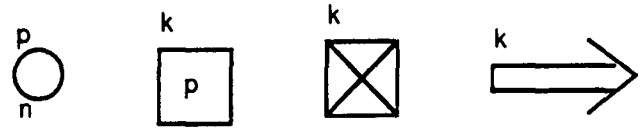


Figure 4.7: Some icons that can be used as input to the Mealy machine.

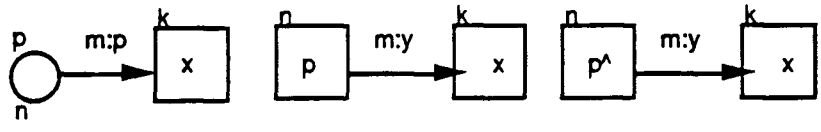


Figure 4.8: Some icons with their pointer assignments that can be used as input to the Mealy machine.

In Figures 4.7 and 4.8, "p" and "x" are the names of the icons, and "p^" is the name of a dereferenced icon. "y" represents a field's name if "p" or "p^" is a record, or an element's name if "p" or "p^" is an array. "n", "m", and "k" are the sequence numbers of the icons.

The output of Table 4.1 (contents of its cells) can set a transformation's type such as (transType := ConditionalTrans), add semantics to Figure 4.5 such as (insert "p=nil" into the "Conditions" list), add a variable to the "Locals" list or the "Globals" list (Declare "p"),

create a new transformation node (Begin a new transformation), change the machine state such as (State := inCondition), or error.

Table 4.1 gives the Mealy machine as a table, and Appendix E gives the machine as a state diagram.

4.5 Code Generator

The input to the code generator is the semantic data structure and the output is a Pascal unit as shown in Figure 4.1.

The code generator copies the "Interface" portion from the semantic data structure to the code's file without modification. Then, it sorts the "Globals" list, "Locals" list, "Conditions" list, and "Actions" list in ascending order by their sequence numbers. Then, it sorts the "Transformations" list in ascending order by their transformation numbers. Then, it traverses the "Globals" list and generates code to declare the global variables as shown in Figure 4.9.

```
var
    name1 : typeName1;
    name2 : typeName2;
      ●
      ●
      ●
```

Figure 4.9: Declarations for the global variables.

In Figure 4.9, "namei", and "typeNamei" correspond to the "name" and "typeName" fields of the i^{th} element in the "Globals" list of Figure 4.5.

Then, the code generator traverses the "Operations" list and performs the following steps. It copies the operation's header ("name" field) and the "Types&Constants" portion without modification. Then it traverses the "Locals" list and generates code to declare the local variables as shown in Figure 4.10.

```
var
    name1 : typeName1;
    name2 : typeName2;
      ●
      ●
      ●
```

Figure 4.10: Declarations for the local variables.

In Figure 4.10, "namei", and "typeNamei" correspond to the "name" and "typeName" fields of the i^{th} element in the "Locals" list of Figure 4.5.

Then, the code generator traverses the "Transformations" list and uses the transformation's type ("transType" field in Figure 4.5), to generate code for each transformation as shown in Table 4.2. The code generator terminates when it reaches the end of the "Operations" list of Figure 4.5. A summary for the code generator is given in Algorithm 2.

Algorithm 2 - Code Generator.

Input: A semantic data structure of Figure 4.5.

Output: A Pascal unit.

Method:

Copy the "Interface" portion of Figure 4.5 into the code's file.

Sort the lists - "Globals", "Locals", "Conditions", "Actions", and "Transformations".

Generate declarations for the "Globals" list.

For all operations in the "Operations" list of Figure 4.5

Copy operation's header and "Types&Constants".

Generate declaration for the "Locals" list.

For all transformations in the "Transformations" list

Generate code according to Table 4.2.

Terminate.

Table 4.2 A transformation's Pascal code.

transType	Pascal Code

ConditionalTrans	if (cond1 and cond2 and ...) then
	begin
	act1
	act2
	.
	.
	.
	end;

FunctionTrans	if (cond1 and cond2 and ...) then function-name := act;

LoopTrans	while (cond1 and cond2 and ...) do
	begin
	act1
	act2
	.
	.
	.
	end;

SequenceTrans	act1 act2 . . .

4.6 Example

Consider the semantics of the binary tree example in Figure 4.6. The code is generated for this module as follows. The code generator copies the "Interface" portion as shown in Figure 4.11.

```
unit    LINKED_LIST;
interface
type
        LL_list   =  ^LL_node;
        LL_node   =  record
                     data   :   string;
                     next   :   LL_list;
                     end;
procedure LL_TRAVERSE(list  :  LL_list);
implementation
```

Figure 4.11: Code for the interface of the LINKED_LIST module.

Then, the code generator sorts the lists in Figure 4.6 and generates code to declare the global variables, which results in no action because the lists are already sorted, and there are no global variables. Then, it copies the operations header as shown in Figure 4.12.

```
unit    LINKED_LIST;
interface
type
        LL_list   =  ^LL_node;
        LL_node   =  record
                     data   :   string;
                     next   :   LL_list;
                     end;
procedure LL_TRAVERSE(list  :  LL_list);
implementation

procedure LL_TRAVERSE;
```

Figure 4.12: Code for the interface and the operation's header of the LINKED_LIST module.

Then, the code generator copies the "Types&Constants" portion and generates declarations for the "Locals" list, which results in no action because there is no "Types&Constants" and there are no local variables. Then, it consults Table 4.2 and generates code for the transformation as shown in Figure 4.13.

```
unit    LINKED_LIST;
interface
type
        LL_list   =   ^LL_node;
        LL_node   =   record
                        data   :   string;
                        next   :   LL_list;
                        end;
procedure LL_TRAVERSE(list : LL_list);
implementation

procedure LL_TRAVERSE;
        begin
                if (list <> nil) then
                    begin
                        writeln(list^.data);
                        LL_TRAVERSE(list^.next);
                    end;
        end;
end.
```

Figure 4.13: Code for the the LINKED_LIST module.

4.7 Summary

In this chapter we present a method for mapping visual/textual specifications into Pascal code. For the current implementation we use Pascal syntax to generate both semantics and code, but the method itself is language-independent. The semantic generator can be used to generate semantics in different languages, by changing the generated semantics. For example, instead of generating

insert "p <> nil" into the "Conditions" list

we can generate the equivalent in C syntax:

insert "p != nil" into the "Conditions" list.

Similarly, The code generator is language-independent. Table 4.3 is similar to Table 4.2, but it generates C code instead of Pascal.

The code generator performs a limited code optimization. For example, the code generated for a "Sequence" transformation in Table 4.2 is more efficient than the actual code shown in Figure 4.14.

```
if true then
  begin
        act1
        act2
          .
          .
          .
  end;
```

Figure 4.14: A non-efficient code for a Sequence transformation.

Further research is needed to generate more efficient codes for all transformations.

Table 4.3: A transformation's C code.

transType	C Code

ConditionalTrans	if (cond1 and cond2 and ...) then
	{
	act1
	act2
	.
	.
	.
	};

FunctionTrans	if (cond1 and cond2 and ...) then return(act);

LoopTrans	while (cond1 and cond2 and ...)
	{
	act1
	act2
	.
	.
	.
	};

SequenceTrans	act1 act2 . . .

Chapter 5

Conclusion and Future Research

5.1 Observations

DataLab has been implemented on the Apple Macintosh and makes full use of that machine's graphics capabilities. It is a testbed for the use of both text and graphics in specifying abstract data types, and for the process of generating imperative code from such specifications. A summary of the observations that we made in designing and implementing DataLab are listed below.

1) Visual representation is useful for data structures, especially dynamic data structures and algorithms. It helps in understanding and maintaining programs.
2) Textual representation is useful for some programming constructs, such as input/output statements, data structure definitions (declarations), and procedure calls.
3) Visual representation of programs allows the user to randomly access any information on the screen.
4) The size of a computer's screen restricts the power of graphics-based programming languages. Visual languages, unlike textual languages, are multidimensional; for the sake of understanding, it is therefore important to see a picture of the whole program on one screen rather than on several screens.

5) Text-based languages have efficient tools for debugging, changing, and maintaining programs. Graphics-based languages need similar tools to make them easier to work with.

6) Text-based languages are more efficient in memory usage than graphics-based languages. Further research is needed to find algorithms to store and retrieve icons in an efficient way.

7) The combination of graphics and an example-based approach provides a powerful and yet easy-to-use technique for representing programs. It helps in understanding and visualizing any program.

8) Successful visual/textual programming languages don't need fancy icons; as a matter of fact, we believe that simplicity is very important in programming languages. For this reason, we use only two icons to represent data structures: a box to represent static data structures and an arrow to represent dynamic data structures.

9) Portability of the generated code is very important. For this reason, DataLab generates target source code (currently Pascal). This makes the generated code portable to other machines and helps to take advantage of programming tools on the other machines, such as debuggers and code optimizers.

5.2 Open Problems

DataLab emphasizes visualization of data structures and enforces encapsulation and modularity of software. We hope that DataLab presents a good example to inspire other researchers in the field of programming languages. However, there are many problems that need to be solved in order for visual/textual languages to gain wide acceptance from the programming community. Some of these problems are listed below.

1) Develop a formal theory for visual/textual programming languages. For example, visual/textual programming languages need something similar to the BNF specification, to parsing techniques, and to optimization algorithms in the textual languages.
2) Develop algorithms for editing and efficiently storing and retrieving visual/textual representations. These algorithms are very important in making visual/textual languages comparable in speed, memory usage, and flexibility to textual languages.
3) Conduct controlled experiments to find the best way of using a combination of text and graphics to represent abstract data types. The representation should produce "fast," "efficient," and "maintainable" programs, and at the same time make the process of programming easier to understand and perform.

5.3 Limitations

The current implementation of DataLab has the limitations listed below.

1) Users should be familiar with abstract data types and modular design concepts.
2) Changes to visual specifications are restricted to enforce consistency between textual and visual specifications. A user is not allowed to delete an icon within a transformation; the user can either delete a complete transformation or delete the last created icon in the last transformation.
3) Changing an interface specification is allowed, but the user is responsible for keeping the consistency between the visual and the textual specifications. If the user changes an interface specification before creating any visual specification, then there is no problem. However, if there are some visual specifications when the changes to the interface

specification is made, then the user must delete all transformations that are affected by the changes and reconstruct them.

4) Array elements and record fields can be dereferenced once, while pointers can be dereferenced twice.
5) It is not allowed to define an ADT's local procedures and functions; all procedures and functions must be listed in the interface part (exported).

5.4 Extensions

In addition to the limitations listed above, some of the extensions of DataLab that need further research are listed below.

5.4.1 Graphical Editor

The graphical editor allows the user to specify the interface part in textual format and the implementation part in a combination of text and graphics format.

A text window is provided by the editor to enter the specification of the interface part textually. For the current implementation of DataLab this specification is limited to Pascal syntax. One direction for future research is to extend the editor to allow language independent specifications of the interface part. This extension requires the development of an internal representation for the interface part and modification of the editor to manipulate this representation. The internal representation must be general to include the components of the interface part, such as imported modules, exported constants, exported types, exported variables, and exported operations (procedures and functions). The editor must be

modified to parse this representation and extract from it the information needed to define the operations, such as the list of imported modules, the list of user-defined types, and the list of operations.

Another direction for future research is to extend the capabilities of the editor to allow the user to modify the specifications of both the interface and the operations, while maintaining the consistency of the specifications after the modifications. Maintaining the consistency is very important, because modifying the interface part requires modifying the associated visual specifications of the operations. For example, modifying a "type" definition in the interface part requires modifying all visual objects of that type that are used to define one or more operations in a particular module.

5.4.2 Source Code Generator

The code generated by DataLab is compilable and can be used in any Pascal program. In fact, DataLab's ADTs are incorporated in applications which are automatically generated (Lewis, 1988). The generator is also limited to generation of Pascal code. One direction for future research is to extend the code generator to map the visual/textual specifications into the code of several programming languages. This extension requires the use of different transformation algorithms for different programming languages. Another approach for this extension is the use of knowledge-based systems, where the knowledge consists of syntax for various programming languages.

5.4.3 Applications

One direction for future research is to investigate the application of DataLab in different domains. The example-based approach makes DataLab suitable for applications such as natural language processing (NLP), and the visualization of data structures makes DataLab a candidate for applications such as teaching data structures and algorithms in computer science classes.

5.4.4 Code Optimization

DataLab performs a limited code optimization, as pointed out in chapter 4. More research is needed to map the visual/textual specifications into an efficient code. For example, a sequence of Conditional transformations is mapped into a sequence of "if-then" statements instead of one "if-then-else" statement. In general, it is not true that any sequence of "if-then" statements is equivalent to one "if-then-else" statement.

5.4.5 DataLab

One of the goals of DataLab is to show that ADTs can be specified and implemented more quickly and reliably through DataLab than by hand. One direction for future research is to conduct controlled experiments to support this conjecture.

5.5 Statistics

1) DataLab application size: 63K.

2) DataLab LightSpeed project size: 237K.

3) Resource size:

   -- Uncompiled (DSD.R): 15K.

   -- Compiled (DSD.RSRC): 6K.

4) Number of units in DataLab: 18.

Bibliography

Al-Mulhem, M. S., Lewis, T. G., "DataLab: A Graphical System for Specifying and Synthesizing Abstract Data Types," Technical Report No. 89-60-9, Oregon State University, Corvallis, OR 97331-4602, June 1989.a.

Al-Mulhem, M. S., Lewis, T. G., "Program Transformation Method in DataLab," Technical Report No. 89-60-14, Oregon State University, Corvallis, OR 97331-4602, July 1989.b.

Al-Mulhem, M. S., Lewis, T. G., "Programming Aspects of DataLab: A Graphical System for Specifying and Synthesizing Abstract Data Types," Technical Report No. 89-60-23, Oregon State University, Corvallis, OR 97331-4602, September 1989.c.

Biermann, A. W., and Krishnaswamy, R., "Constructing Program From Example Computations", IEEE Transaction on Software Engineering, Vol. SE-2, No.3, pp141-153, Sept. 1976.

Biermann, A. W.,"The Inference of Regular LISP Programs from Examples", IEEE Transaction on Systems, Man and Cybernetics, Vol. SMC-8, No.8, pp585-600, August 1978.

Borning, A. H., "The Programming Language Aspects Of ThingLab, a Constraint-Oriented Simulation Laboratory," ACM Transaction On Programming Languages and Systems, Vol.3, No. 4, pp353-387, Oct. 1981.

Brooks, R. A., Programming in Common LISP, John Wiley & Sons, 1985.

Brown, G. P., Carling, R. T., Herot, C. F., Kramlich, D. A., and Souza, P., "Program Visualization: Graphical Support for Software Development", IEEE Computer, Vol. 18, No. 8, pp27-37, August 1985.

Brown, M. H., and Sedgewick, R., " Techniques for Algorithm Animation", IEEE Software, Vol. 2, No. 1, pp28-39, Jan. 1985.

Budd, T. , An APL Compiler, Springer-Verlag, New York, 1988.

Chang, S., Ichikawa, T., and Ligomenides, P., Visual Languages, Plenum, New York, 1986.

Chang, S. C.,"Visual Languages : A Tutorial and Survey", IEEE Software, Vol. 4, No. 1, pp29-39, Jan. 1987.

Curry, G. A., Programming by Abstract Demonstration., Ph.D. Dissertation, University of Washington, Seattle, Technical Report # 78-03-02, March 1978.

Dahl, O., and Nygaard, K., "SIMULA an Algol-Based Simulation Language", Comm. ACM 9, 9, pp671-678, 1966.

Dromey, R. G.,"Derivation of Sorting Algorithms from a Single Specification", The Computer Journal, Vol. 30, No. 6, pp512-518, 1987.

Finzer, W., and Gould, L., " Programming by Rehearsal.", Technical Report No. SCL-84-1, Xerox Corporation, Palo Alto Research Center, Palo Alto, California 94304, May 1984.

Foley, J., Kim, W. C., Kovacevic, S., and Murray, K., "Defining Interfaces at a High Level of Abstraction", IEEE Software, pp25-32, Jan. 1989.

Gehani, N., " Specifications : Formal and Informal- a Case Study", Software Practice and Experience, Vol. 12, No. 5, pp433-444, May 1982.

Glinert, E. P., and Tanimoto, S. L., "PICT: An Interactive Graphical Programming Environment", IEEE Computer, Vol. 17, No. 11, pp7-25, Nov. 1984.

Glinert, E. P.,"Interactive, Graphical Programming Environments : Six Open Problems and a Possible Partial Solution", IEEE Computer Society's Tenth Annual International Computer Software and Applications Conference, pp408-410, Oct. 8-10, 1986

Glinert, E. P.,"Out of Flatland: Towards 3-D Visual Programming", IEEE Proceedings 1987 fall Joint Computer Conference Exploring Technology: Today and Tomorrow, pp392-399, Oct. 25-29, 1987.

Grogono, P., Programming in Pascal, Addison-Wesley, California, 1980.

Halbert, D. C., Programming by Example, Ph.D. Dissertation, Computer Science Division, University of California, Berkeley, 1984.

Hix, D., "User Interfaces : Opening a Window on the Computer", IEEE Software, pp8-10, Jan. 1989.

Hopcroft, J. E., and Ullman, J. D., Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, California, 1979.

Hsieh, C., A Graphical Editor for Pascal Programming on Macintosh, Technical Report No. 88-60-4, Oregon State University, Corvallis, Or 97331-4602, 1988.

Ishida, H., and Ohnishi, M., " DF Language, a Program Development Tool Applying The Concept of Data Flow", in Application Development Systems, Edited by Tosiyasu L. Kumi, Springer-Verlag, Tokyo, pp52-64, 1986.

Isoda, S., Shimomura, T., and Ono, Y., " VIPS : A Visual Debugger", IEEE Software, pp8-19, May 1987.

Jalote, P., "Synthesizing Implementations of Abstract Data Types From Axiomatic Specifications", Software Practice and Experience, Vol. 17, No. 11, pp847-858, Nov.1987.

Kim, H., "GEDL: A Graphical Editor for DataLab", Oregon State University, Corvallis, OR 97331-4602, December 1989.

Kodosky, J., and Dye R., "Programming With Pictures," Computer Languages, Vol. 6, No. 1, pp61-69, Jan. 1989.

Lampson, B. W., et.all.,"Report on the programming language Euclid", ACM Sigplan Notices, Vol. 12, No. 2, 1977.

Lewis, T. G., Handloser III, F., Bose, S., and Yang, S.,"Prototypes From Standard User Interface Mangement Systems", Computer Science Dept., Technical Report # 88-60-10, Oregon State University, Corvallis, OR 97331-4602, 1988.

Lieberman, H., "Seeing What Your Programs Are Doing," International Journal Of Man-Machine Studies, Vol.21, No. 4, pp311-331, Oct. 1984.

Martin, J., and Leben, J., Fourth Generation Languages, Volume II, Representative 4GLs, Prentice-Hall , New Jersey 1986.

Monlar, L., Navrat, P., and Vojtek, V.,"A System for Automatic Implementation of Abstract Data Types", Computer and Artificial Intelligence, Vol. 6, No. 1, pp51-58, 1987.

Moriconi, M., and Hare, D. F., "The PegaSys System: Pictures as Formal Documentation of Large Programs", ACM Transaction on Programming Languages and Systems, Vol.8, no. 4, pp524-546, Oct. 1986.

Moriconi, M., and Hare, D. F., "Visualizing Program Designs Through PegaSys", IEEE Computer, Vol. 18, No. 8, pp72-86, August 1985.

Myers, B. A., and Buxton, W.,"Creating Highly-Interactive and Graphical User Interfaces by Demonstration.", SIGGRAPH 86 Proceedings, pp249-258, Aug.1986.

Myers, B. A.,"INCENSE : A system for Displaying Data Structures", Computer Graphics, Vol. 17, No. 3, pp115-125, July 1983.

Polivka, R., and Pakin, S., APL : The Language and its Usage, Prentice-Hall, 1975.

Powell, M., and Linton, M., "Visual Interaction in an Interactive Programming Environment", ACM SIGPLAN in Sigplan Notes, Vol. 18, No. 6, June 1983.

Pratsch, H., and Steinbruggen, R., "Program Transformation Systems," ACM Computer Surveys, Vol. 15, No. 3, pp199-236, Sept. 1983.

Raeder, G., "A Survey of Current Graphical Programming Techniques", IEEE Computer, Vol. 18, No. 8, pp11-25, Aug. 1985.

Raeder, G., Programming in Pictures., Ph.D. Dissertation, Department of Computer Science, University of Southern California, Nov. 1984.

Reiss, S. P., "Graphical Program Development with PECAN Program Development System", ACM SIGPLAN in Sigplan Notices, Vol. 19, No. 5, pp30-41, May 1984.

Reiss, S. P., "Working in the Garden Environment for Conceptual Programming", IEEE Software, Vol. 4, No. 6, pp16-27, November 1987.

Rubin, R. V., Golin, E.J. and Reiss, S. P.,"ThinkPad : a Graphical System for Programming by Demonstration", IEEE Software, Vol. 2, No.2, pp73-79, March 1985.

Rubin, R. V., Reiss, S. P., and Golin, E.J.,"Compiler Aspects of an Environment for Programming by Demonstration", Lecture Notes in Computer Science # 282, Edited by Gorny, P. and Tanber, M. J., pp199-210, 1986.

Schwartz, J. T., Dewar, R. B. K., Dubinsky, E. and Schnoberg, E., Programming with sets: an introduction to SETL, Springer-Verlag, NY 1986.

Shu, N. C., Visual programming, Van Nostrand Reinhold Company, NY 1988.

Shu, N. C.,"A Form-Oriented and Visual Directed Application Development System For Non Programmer," in Application Development Systems, Edited by Tosiyasu L. Kumii, Springer-Verlag, Tokyo, pp2-26,1986.

Smith, D. C., Irby, C., Kimball, R., and Harslen, E., "Designing the Star Interface.", Byte Vol. 7, No. 4, pp242-282, April 1982.

Smith, D. C.,Pygmalion : a Computer Program to Model and Simulate Creative Thought. Ph.D. Dissertation, Stanford University, 1975.

Think's LightSpeed Pascal user's guide and reference manual, THINK Technologies, Inc., 1988.

Wirth, N., Programming in Modula-2, Third, Corrected Edition, Springer-Verlag, New York, 1985.

Wood, R. J., Computer Aided Program Synthesis, Ph.D. Dissertation, University of Maryland, 1982.

Appendices

Appendix A

Icons of DataLab

1. Selection arrow.

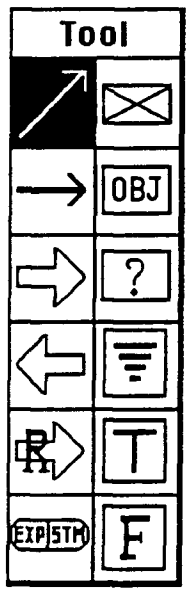
2. Pointer icon.

3. Transform icon.

4. Loop icon.

5. Return icon.

6. Expression/Statement icon



7. Start icon.

8. Object icon.

9. Don't care icon.

10. Nil icon.

11. True icon.

12. False icon.

Appendix B

Objects and Pointers in DataLab

1) Objects

An object's icon is shown in Figure B.1.

Figure B.1: An object's icon.

An object is a five-tuple.

object = (name, type, kind, sequence, OID)

where

name-- This is the object's name. It is defined by the user and can be any legal Pascal identifier. The name is displayed inside the icon and is truncated if it does not fit. Object's name can be changed if the object is a static variable or parameter, by double clicking on the object and clicking on "Change name" button.

type-- This is the object's type. It is defined by the user by selecting from a list of built-in and user defined types (type display dialog). Type information is displayed by double clicking on the object.

kind-- This is the object's kind which can be a local variable, a global variable, or a procedure/function parameter. The user specifies the kind by selecting a radio button in the "object kind dialog". Object's kind is displayed by double clicking on the object.

sequence-- This is the object's sequence number. It corresponds to the creation order of the object in the "Condition/Action" transformation. This number is created automatically by DataLab and is displayed as part of the object representation.

OID-- This is the object's unique ID number. It is assigned to each object when it is created.

2) Pointers

A Pointer's icon is shown in Figure B.2.

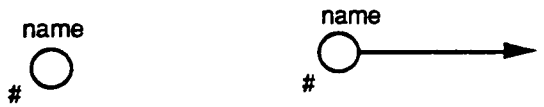


Figure B.2: A pointer's icon: 1) a non-assigned pointer, and 2) an assigned pointer.

Initially, there will be no "arrow" but only the circle. The circle has the same meaning as the object's icon on the screen -- represent memory location for an address or a dereferenced space. The user can move the location of a pointer by dragging the circle. The arrow is used to assign the pointer to a value.

A pointer object is also a five-tuple,

pointer = (name, type, kind, sequence, OID)

where "name", "type", "kind", "sequence" are exactly as in the object's icon. The OID stores the OID of the derefereced object.

Appendix C

Specification Data Structure

```
{-------------------------------------------------------------}
{This unit contains ONLY the specification data structure.}
{-------------------------------------------------------------}

unit UNIT_TYPE_DSD;
interface
uses
   DIALOG_DSD;
type
   UNT_objectKind = (drag, pointerDrag, transform, loop, return, statement, start,
staticObject, undefined, nilValue, trueValue, falseValue, pointer, dynamicObject);

   {Unit part-- refers to a unit frame or an operation}
   UNT_part = (unitFrame, procOperation, funcOperation);

   {Definitions of handles}
   UNT_pointerH = ^UNT_pointerP;
   UNT_pointerP = ^UNT_pointerRecord;
   UNT_objectH = ^UNT_objectP;
   UNT_objectP = ^UNT_objectRecord;
```

```
UNT_unitH = ^UNT_unitP;
UNT_unitP = ^UNT_unitRecord;

{A node in the "Pointer Assignments" list of Figure 4.2 }
UNT_pointerRecord = record
    {general pointer information}
    name : Str255;              {pointer name}
    typeName : Str255;          {type name -used for checking.}
    pointedOID : longint;       {OID for th pointed to icon.}
    order : Str255;

    {drawing information}
    drawingRect : Rect;
    positionRect : Rect;
    startPoint, endPoint : Point;
    highlight : boolean;
    copy : boolean;

    {handles to object and next pointer}
    pointTo : UNT_objectH;
    next : UNT_pointerH;
end;

{A node in the "Objects" list of Figure 4.2.}}
UNT_objectRecord = record
```

```
    {general icon information}
    name : Str255;                  {icon's name}
    typeName : Str255;              {icon's type name}
    OID : longint;                  {icon'sOID }
    kind : UNT_objectKind;          {icon's kind}
    scope : DIA_objectScope;        {icon's scope}}
    order : Str255;                 {icon's sequence number}
    isArray : boolean;

    {drawing information}
    drawingRect : Rect;
    positionRect : Rect;
    highlight : boolean;
    copy : boolean;

    pointerH : UNT_pointerH;
    next : UNT_objectH;
end;

{Unit record definition}
UNT_unitRecord = record
    name : Str255;              {unit name or unit operation's name}
    part : UNT_part;            {unit frame or operation}
    objectH : UNT_objectH;      {an operation's graphical representation.}
    endObject : UNT_objectH;    {points to the last icon}
```

```
        objectOrder : longint;          {maintains a last order of the operation}
        textH : Handle;                 {an operation's local type or unit interface info.}
        next : UNT_unitH;
    end;}

implementation
        {Nothing here}
end.
```

Appendix D

Semantic Data Structure

```
{-------------------------------------------------------------------------------}
{This unit contains the semantic data structure. It also contains global variables}
{used throught the application.}
{-------------------------------------------------------------------------------}

unit GLOBAL_SCG;
interface
type

    {operation kinds}
    GLO_operationKind = (proc, func);

    {transformation types}
    GLO_transTypes = (ConditionalTrans, FunctionTrans, LoopTrans, SequenceTrans);

    {variable definition}
    GLO_varH = ^GLO_varP;
    GLO_varP = ^GLO_varRecord;
    GLO_varRecord = record
        order : integer;                {sequence number.}
```

```
    name: str255;              {variable's name.}
    typeName: str255;          {type's name of the variable.}
    next: GLO_varH;            {next variable.}
end;

{semantic definition.}
GLO_codeH = ^GLO_codeP;
GLO_codeP = ^GLO_codeRecord;
GLO_codeRecord = record
    order : integer;           {sequence number.}
    code: str255;              {semantics in a language syntax(code)}
    next: GLO_codeH;           {next semantic.}
end;

{transformation definition.}
GLO_transH = ^GLO_transP;
GLO_transP = ^GLO_transRecord;
GLO_transRecord = record
    order : str255;                        {case number.}
    transType : GLO_transType;             {transformation's type.}
    conditions, actions : GLO_codeH;       {conditions and actions lists.}
    next : GLO_transH;                     {next transformation.}
end;
```

```
    {operation code definition.}
    GLO_opCodeH = ^GLO_opCodep;
    GLO_opCodeP = ^GLO_opCodeRecord;
    GLO_opCodeRecord = record
        name : str255;                  {operation's name.}
        opKind : GLO_operationKind;     {operation's kind.}
        transformations : GLO_transH;   {transformations list.}
        locals : GLO_varH;              {local variables list.}
        typeConst : Handle;             {local types and constants list.}
        next : GLO_opCodeH;             {next operation.}
    end;

    {unit representation}
    GLO_semH = ^GLO_semP;
    GLO_semP = ^GLO_semRecord;
    GLO_semRecord = record
        globalVars : GLO_varH;          {global variables list.}
        interfacePart : Handle;         {unit's interface.}
        operations : GLO_opCodeH;       {operations list.}
    end;
implementation
end.                                    {unit GLOBAL_SCG}
```

Appendix E

The Mealy Machine

The Mealy machine for the semantic generator is shown as a state diagram in Figures E.1 - E.12.

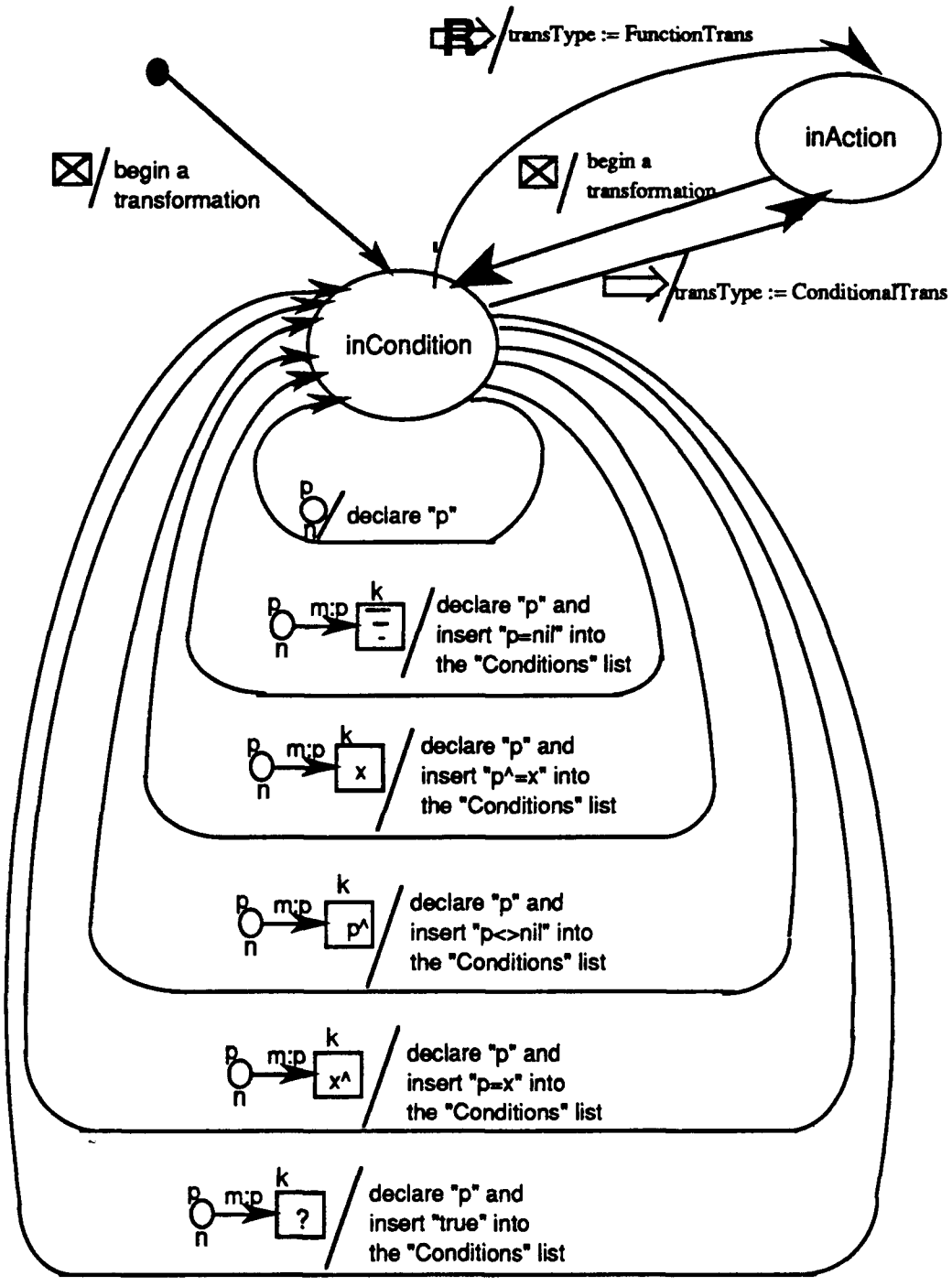


Figure E.1: Mealy machine for "start", "transform", and "return" icons. Also, it handles icons of type "pointer" in the "inCondition" state.

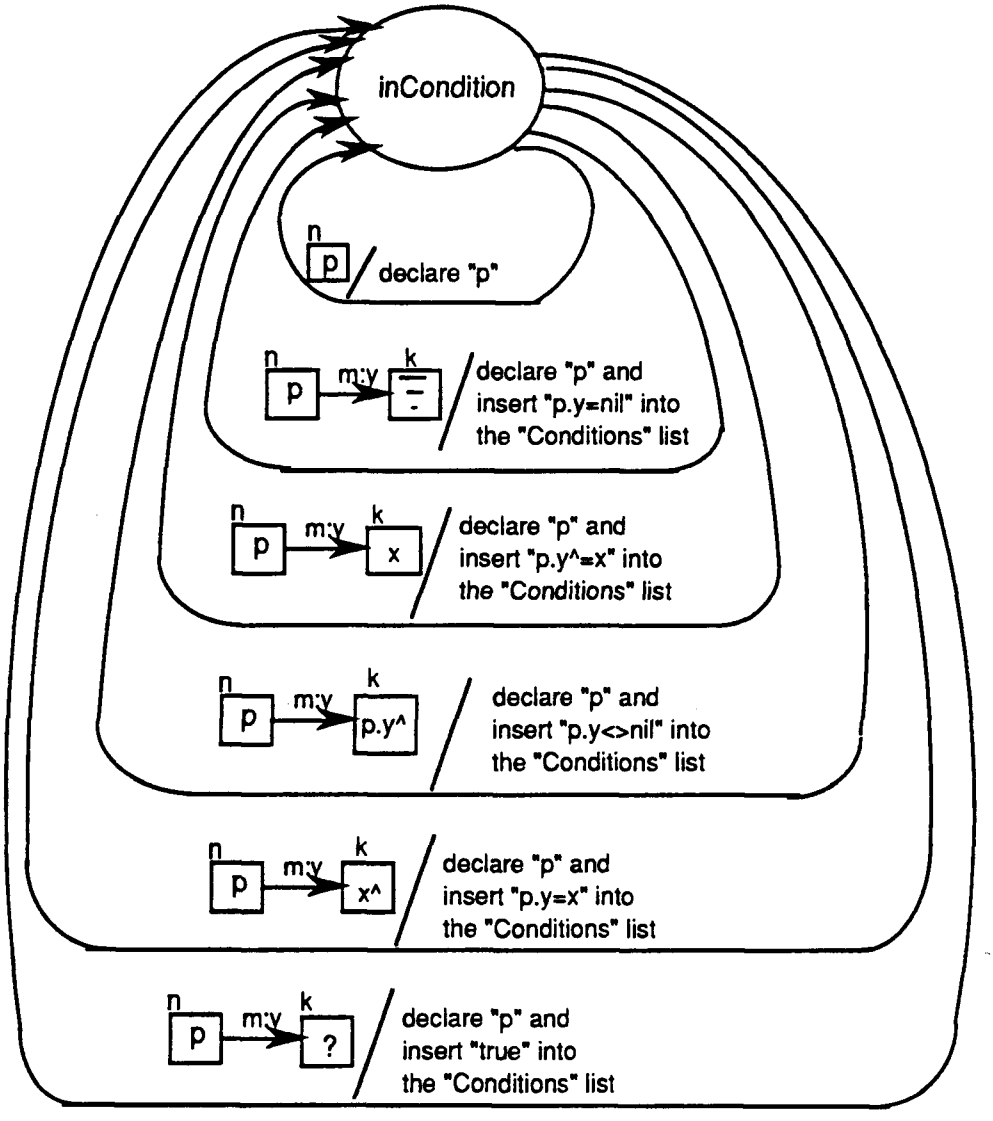


Figure E.2: Mealy machine for an icon of kind "staticObject" and type "record" in the "inCondition" state.

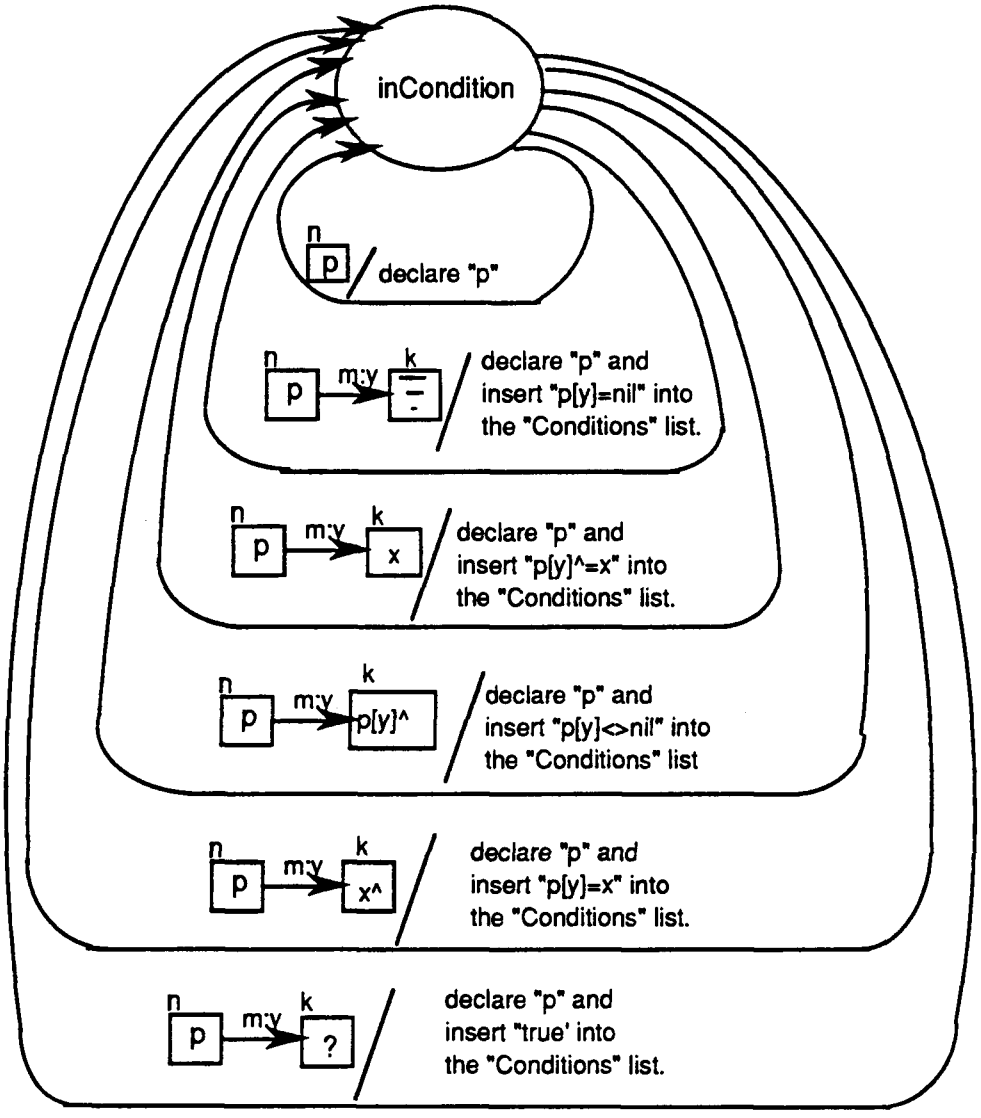


Figure E.3: Mealy machine for an icon of kind "staticObject" and type "array" in the "inCondition" state.

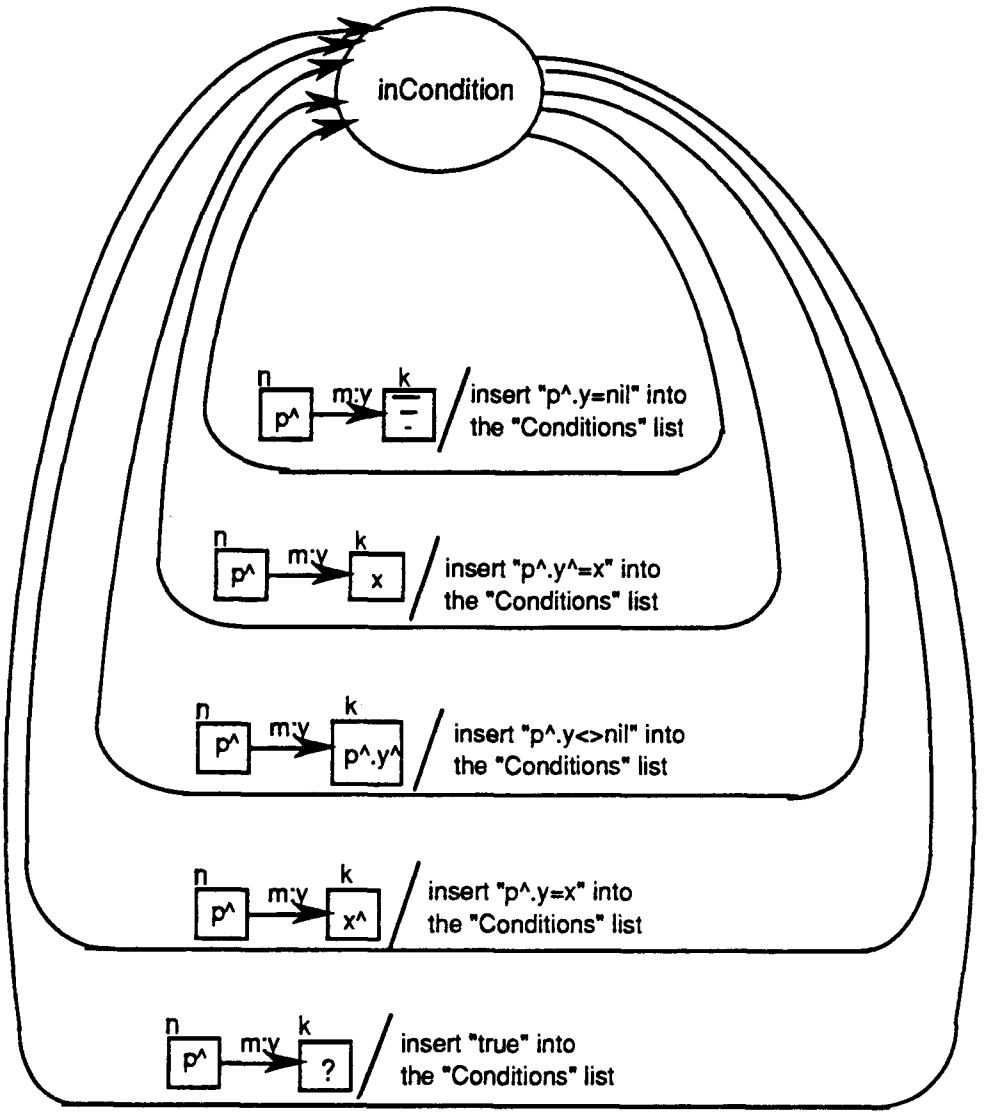


Figure E.4: Mealy machine for an icon of kind "dynamicObject" and type "record" in the "inCondition" state.

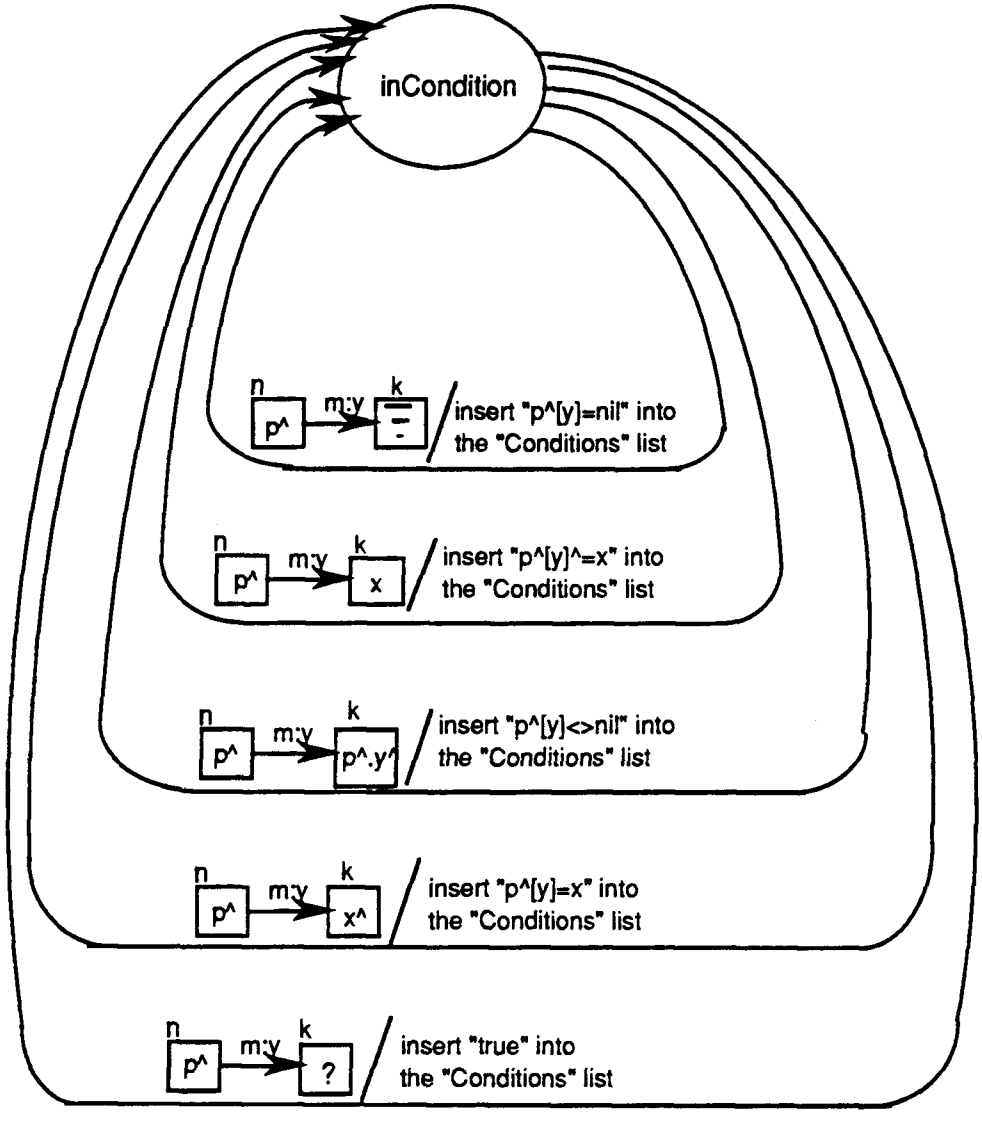


Figure E.5: Mealy machine for an icon of kind "dynamicObject" and type "array" in the "inCondition" state.

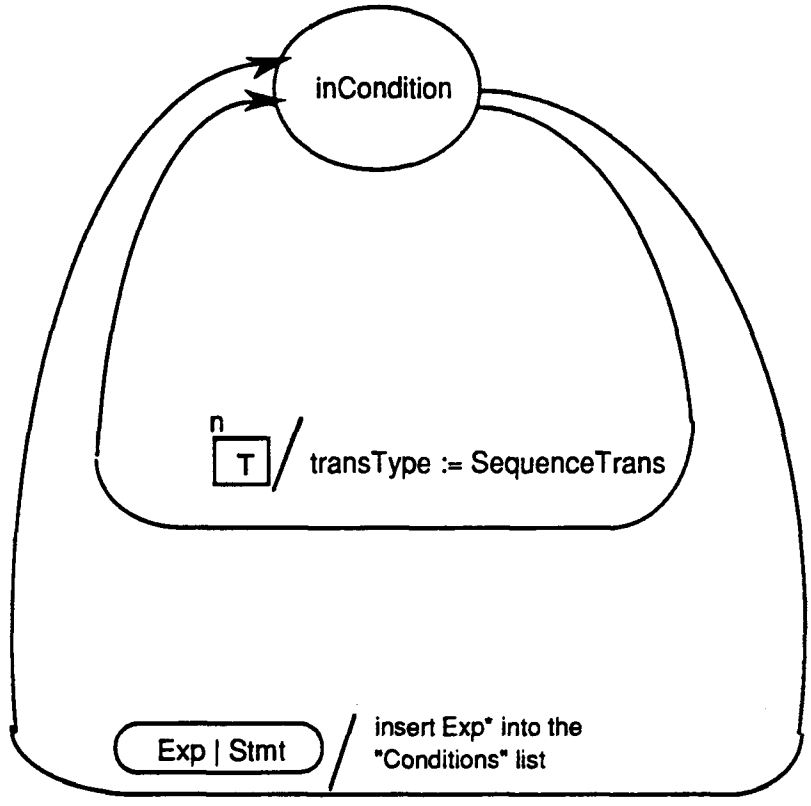


Figure E.6: Mealy machine for icons of kind "trueValue", and "statement" in the "inCondition" state.

* Exp is the expression entered by the user and contained in the "Exp|Stmt" icon.

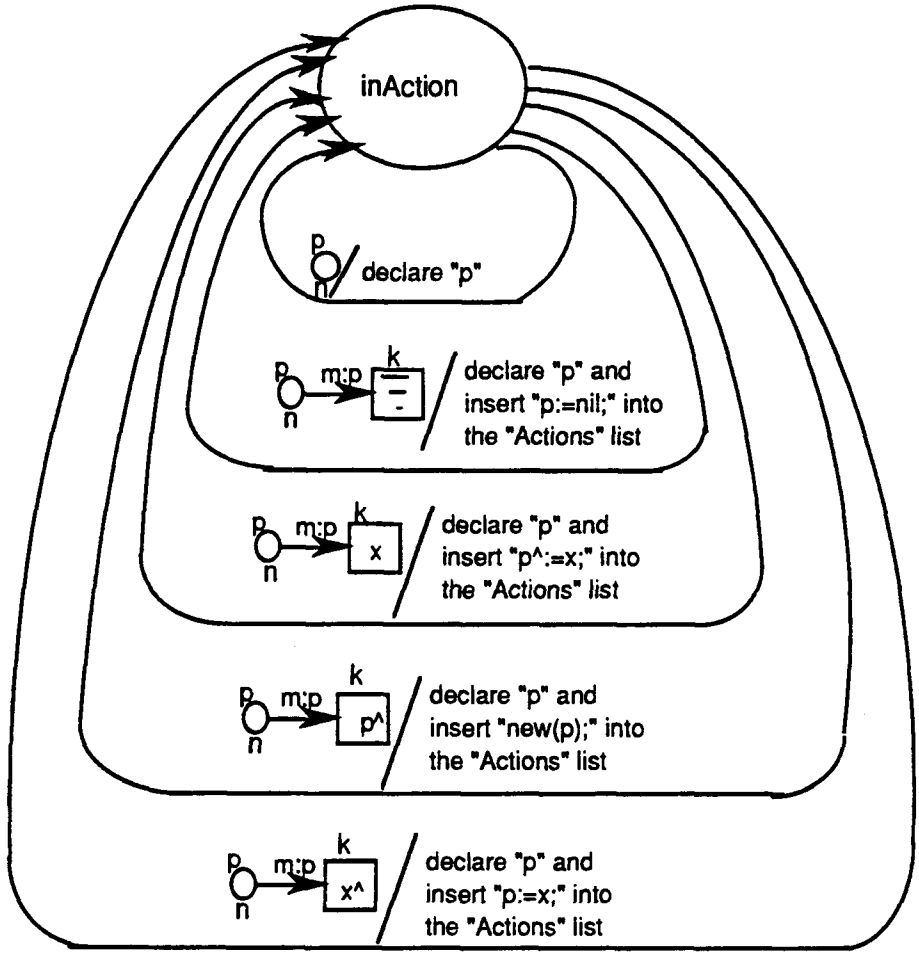


Figure E.7: Mealy machine for icons of kind "pointer" in the "inAction" state.

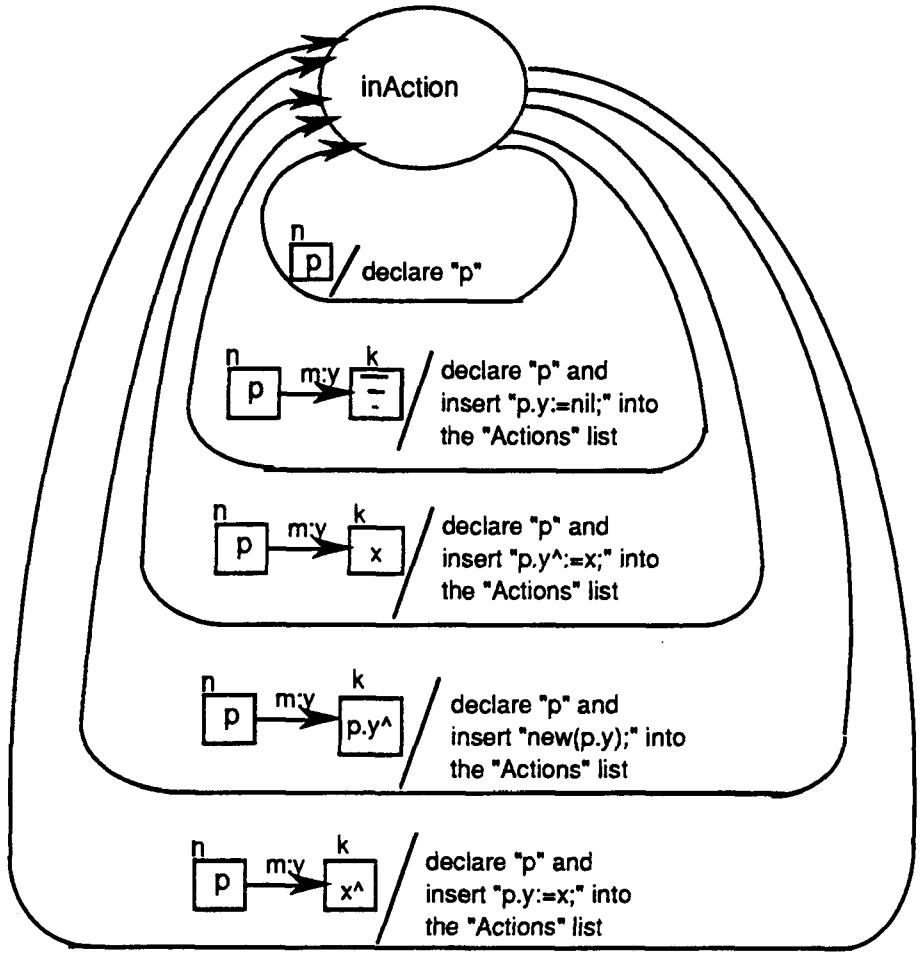


Figure E.8: Mealy machine for an icon of kind "staticObject" and type "record" in the "inAction" state.

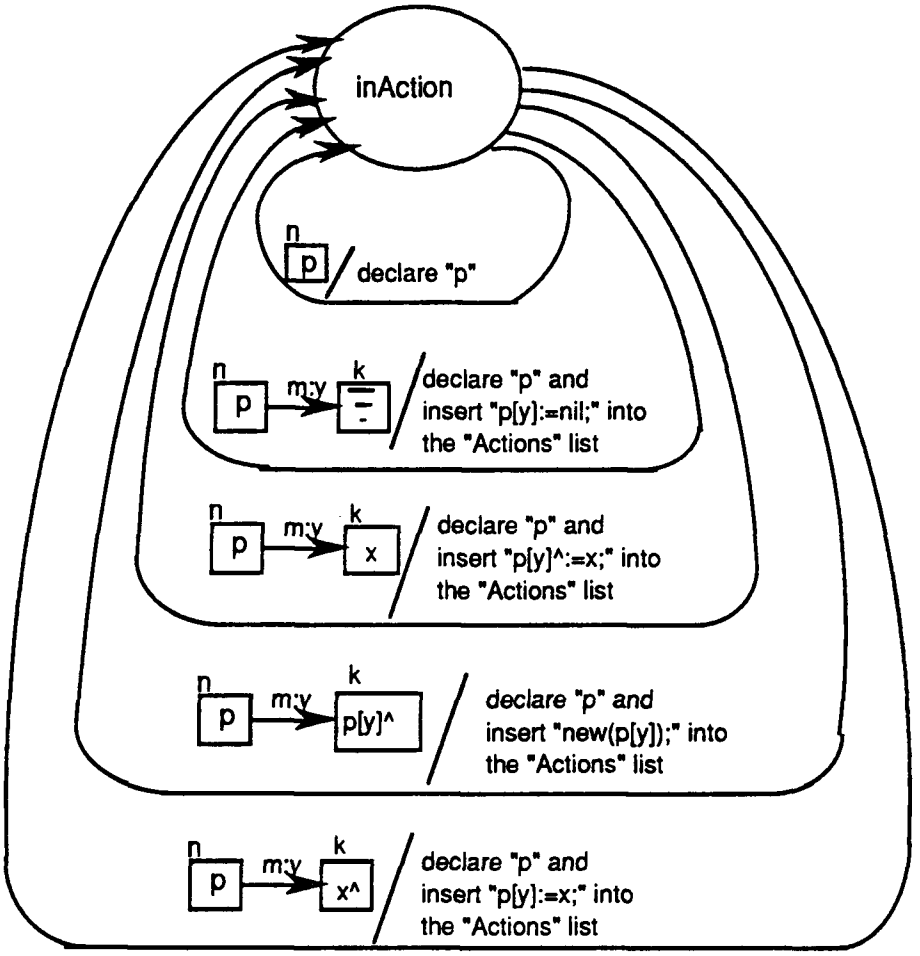


Figure E.9: Mealy machine for an icon of kind "staticObject" and type "array" in the "inAction" state.

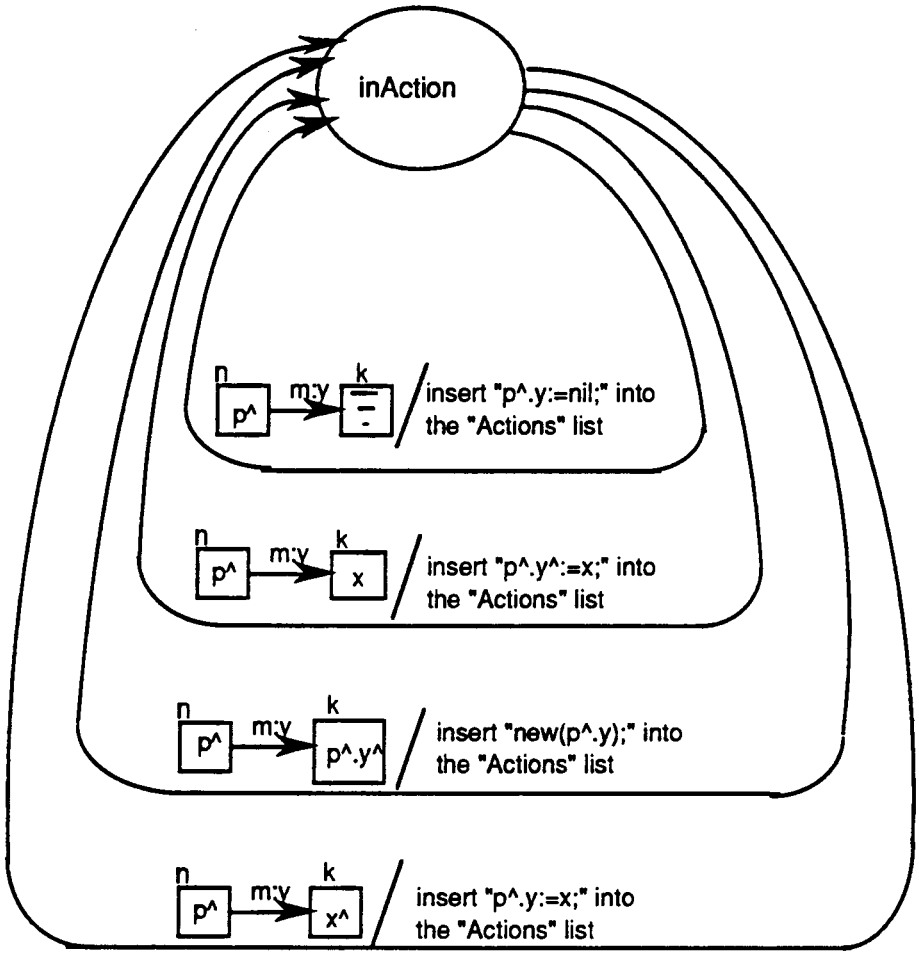


Figure E.10: Mealy machine for an icon of kind "dynamicObject" and type "record" in the "inAction" state.

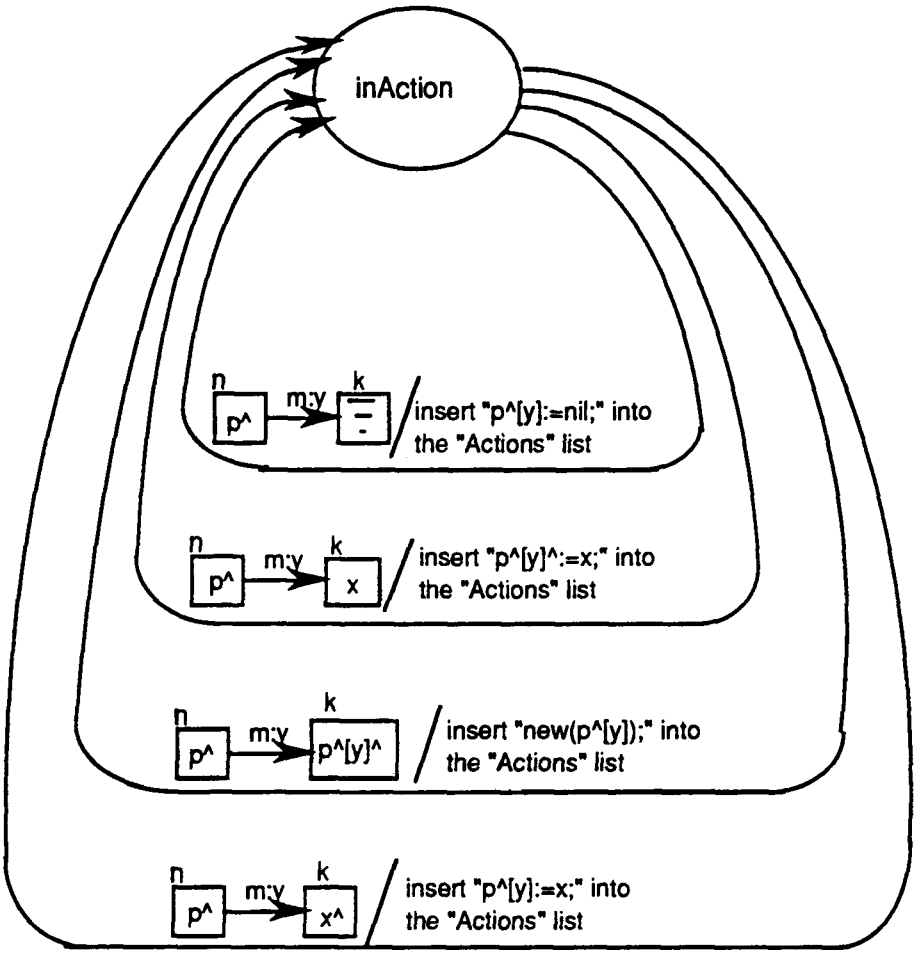


Figure E.11: Mealy machine for an icon of kind "dynamicObject" and type "array" in the "inAction" state.

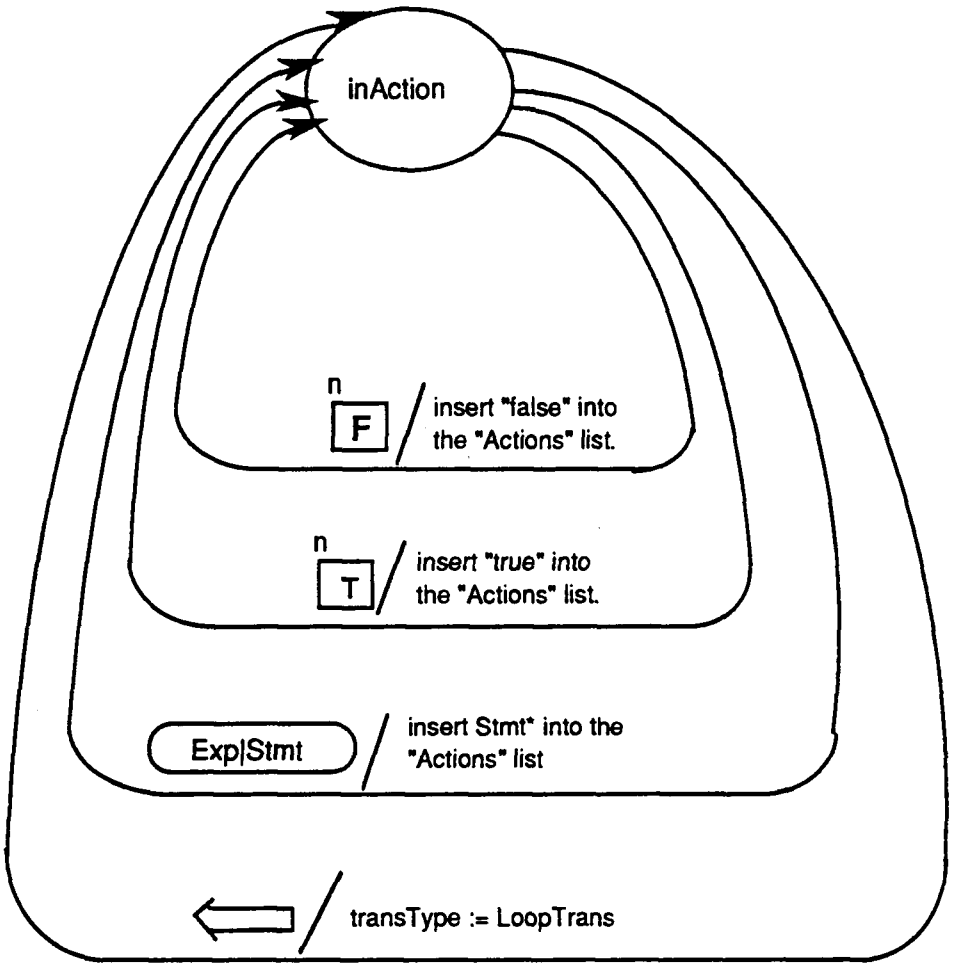


Figure E.12: Mealy machine for icons of kind "nilValue", "trueValue", "falseValue", "loop", and "statement" in the "inAction" state.

*Stmt is the statement entered by the user and contained in the "Exp|Stmt" icon.

Appendix F

Examples

This appendix gives five complete examples for the following fully encapsulated ADTs: 1) Hash Table, 2) FIFO Queue, 3) Single Linked List with sorting operation, 4) Doubly Linked List, and 5) Binary Tree. It shows the DataLab specifications and the automatically generated Pascal code for each example. The generated code for all of the examples in this appendix has been tested using LightSpeed Pascal version 2.0. Each example has been built as a project and a test driver has been used to test them on real data. The test drivers and the results of running theses examples are also shown.

1) Hash Table

This example implements a hash table with four operations: HT_INITIALIZE, HT_INSERT, HT_DELETE, and HT_PRINT. It uses chaining to resolve collisions; therefore, the hash table is implemented as an array of linked lists. HT_INITIALIZE initializes the hash table, HT_INSERT inserts an integer into the proper table entry (bucket), HT_DELETE deletes an integer from the table, and HT_PRINT prints the contents of the table.. The specifications for this module, the generated code, the test driver, and the test results are shown in Figures F.1.1-F.1.8.

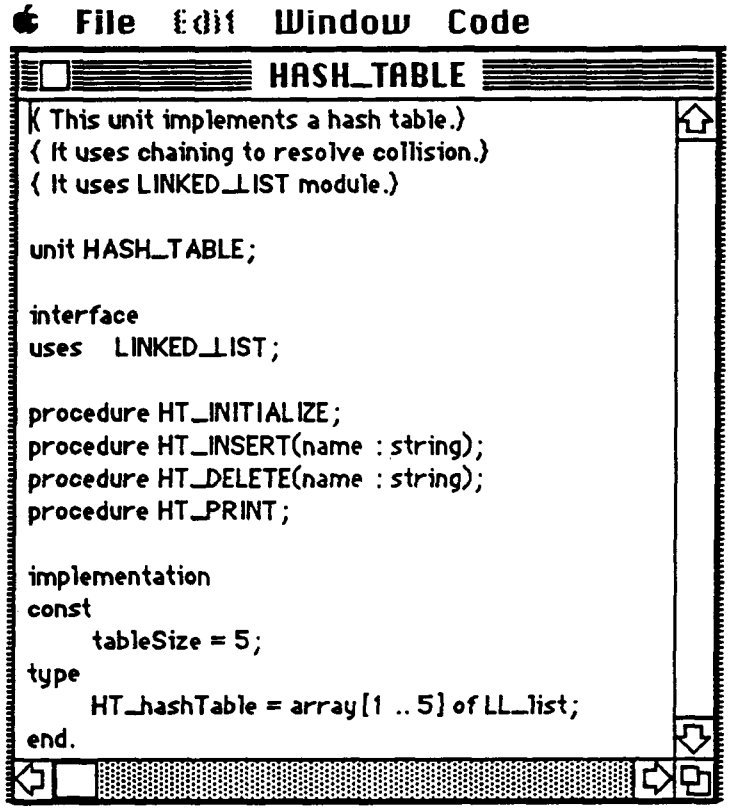


Figure F.1.1: Interface for the hash table example.

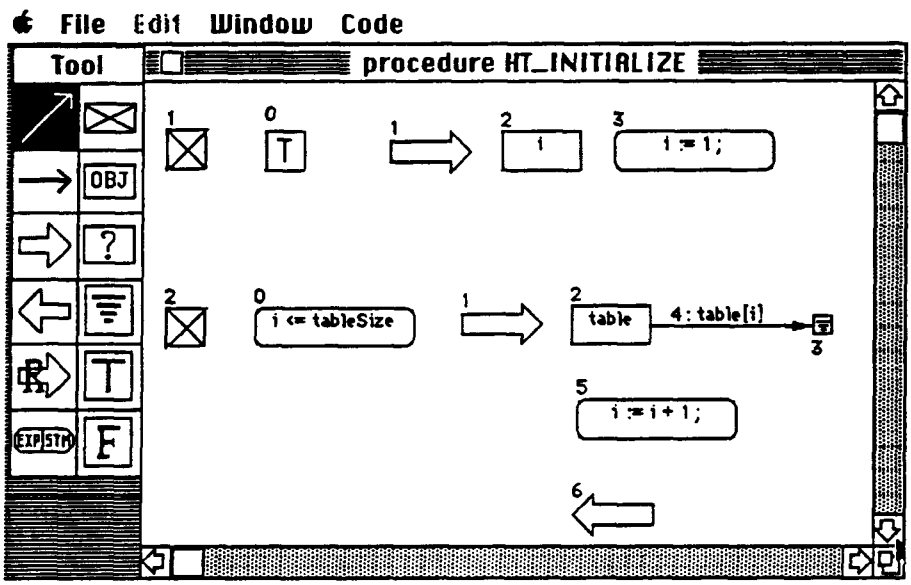


Figure F.1.2: HT_INITIALIZE operation for the hash table example.

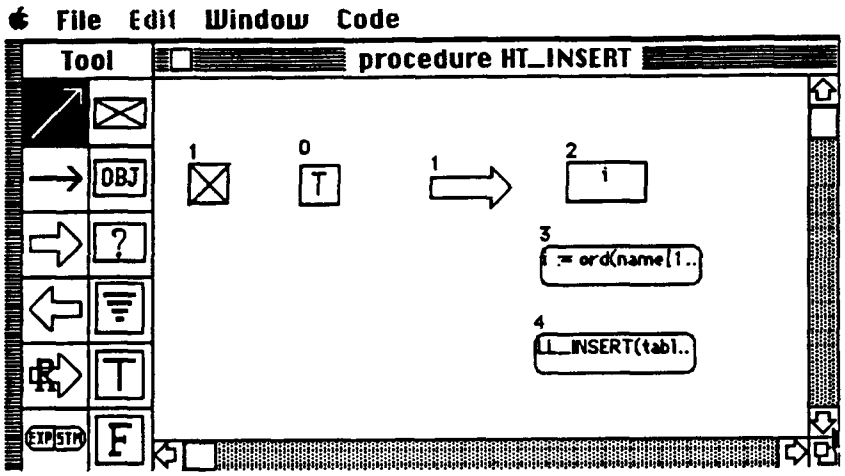


Figure F.1.3: HT_INSERT operation for the hash table example.

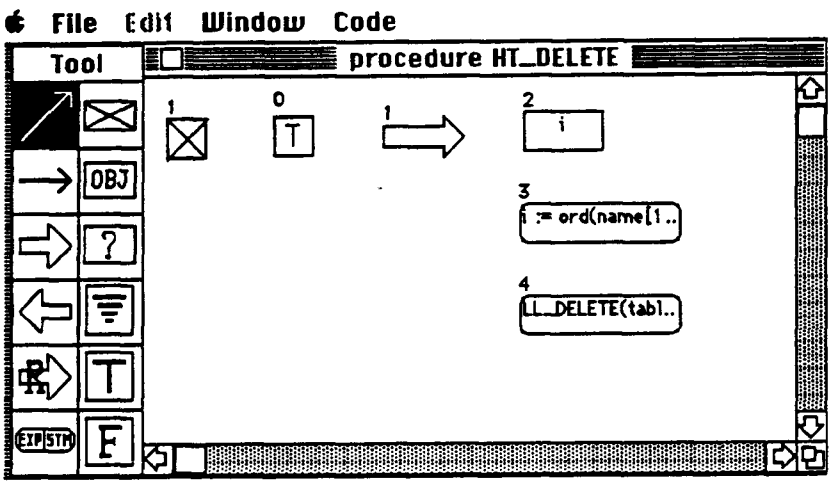


Figure F.1.4: HT_DELETE operation for the hash table example.

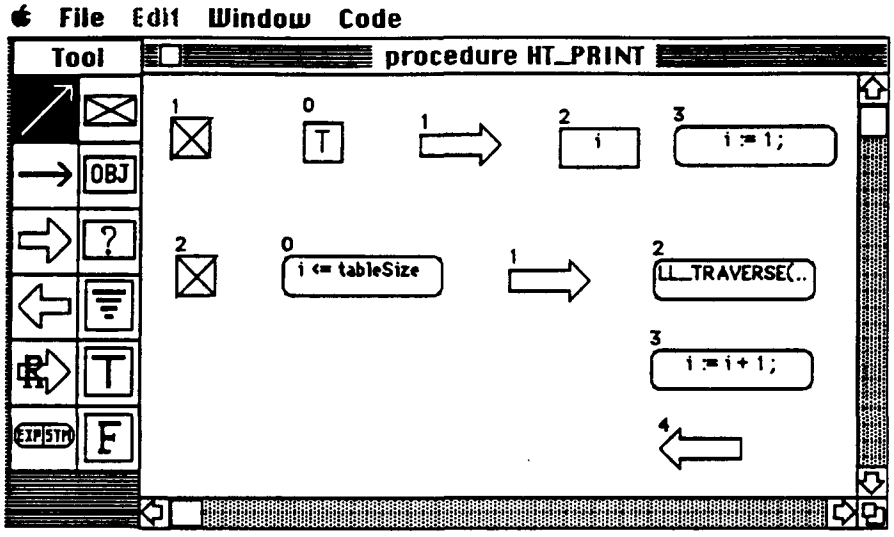


Figure F.1.5: HT_PRINT operation for the hash table example.

```
{ This unit implements a hash table.}
{ It uses chaining to resolve collision.}
{ It uses LINKED_LIST module.}

unit HASH_TABLE;

interface
 uses
  LINKED_LIST;

 procedure HT_INITIALIZE;
 procedure HT_INSERT (name: string);
 procedure HT_DELETE (name: string);
 procedure HT_PRINT;

implementation
 const
  tableSize = 5;
 type
  HT_hashTable = array[1..5] of LL_list;

 var
  table: HT_hashTable;
 procedure HT_INITIALIZE;
  var
   i: INTEGER;
 begin {procedure HT_INITIALIZE}
  i := 1;
  while (i <= tableSize) do
   begin
    table[i] := nil;
    i := i + 1;
   end;
 end; {procedure HT_INITIALIZE}
 procedure HT_INSERT;
  var
   i: INTEGER;
 begin {procedure HT_INSERT}
  i := ord(name[1]) mod tableSize + 1;
  LL_INSERT(table[i], name);
 end; {procedure HT_INSERT}
 procedure HT_DELETE;
  var
   i: INTEGER;
 begin {procedure HT_DELETE}
  i := ord(name[1]) mod tableSize + 1;
  LL_DELETE(table[i], name);
 end; {procedure HT_DELETE}
 procedure HT_PRINT;
  var
   i: INTEGER;
 begin {procedure HT_PRINT}
  i := 1;
  while (i <= tableSize) do
   begin
    LL_TRAVERSE(table[i]);
    i := i + 1;
   end;
 end; {procedure HT_PRINT}
end.
```

Figure F.1.6: The generated Pascal code for the hash table example.

```
program hashTable;
 uses
  HASH_TABLE;
begin
 showText;
 HT_INITIALIZE;
 HT_INSERT('Kid');
 HT_INSERT('Tom');
 HT_INSERT('Dave');
 HT_INSERT('Mary');
 HT_INSERT('Bob');
 HT_INSERT('Gorge');
 writeln('The names in the hash table are:');
 HT_PRINT;
 HT_DELETE('Bob');
 writeln;
 writeln('The names After deleting Bob are:');
 HT_PRINT;
end.
```

Figure F.1.7: The test driver for the hash table example.

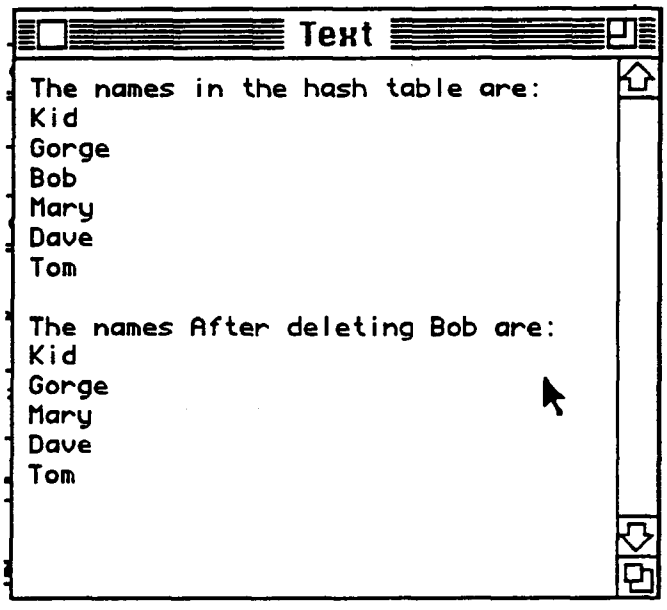


Figure F.1.8: The test results for the hash table example.

2) FIFO Queue

This example implements a queue of integers, as shown in Figure F.2.1. It has four operations: QU_CREATE to create an instance of the queue, QU_DELETE to delete the integer at the front of the queue, QU_INSERT to insert the integer at the rear of the queue, and QU_TRAVERSE to print the integers in the queue. The specifications of this module, the generated code, the test driver, and the test results are shown in Figures F.2.2-F.2.9.

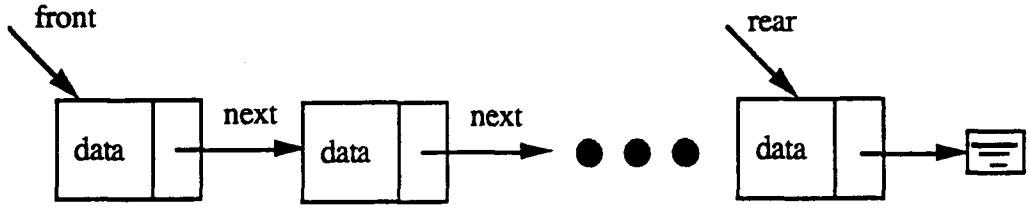


Figure F.2.1:A queue data structure.

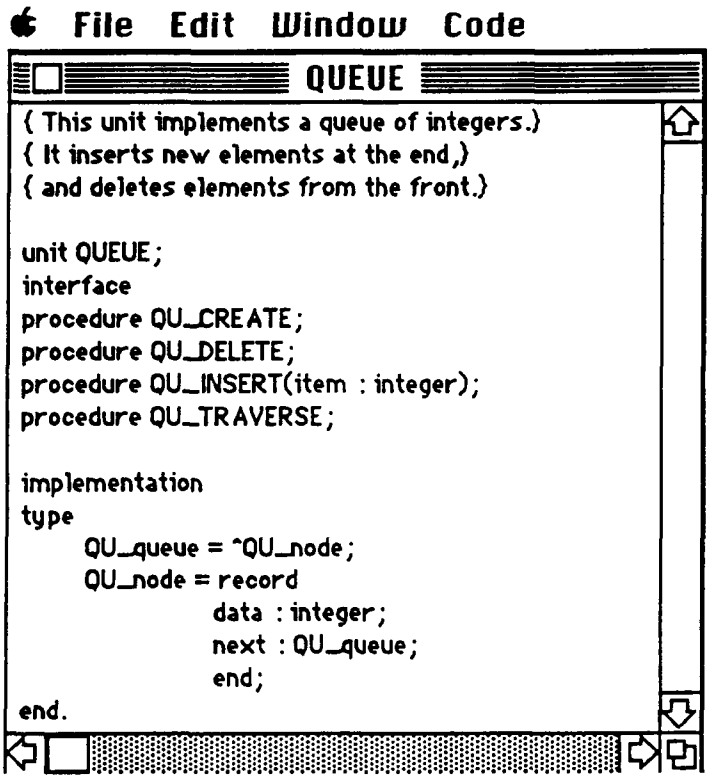


Figure F.2.2: Interface for the queue example.

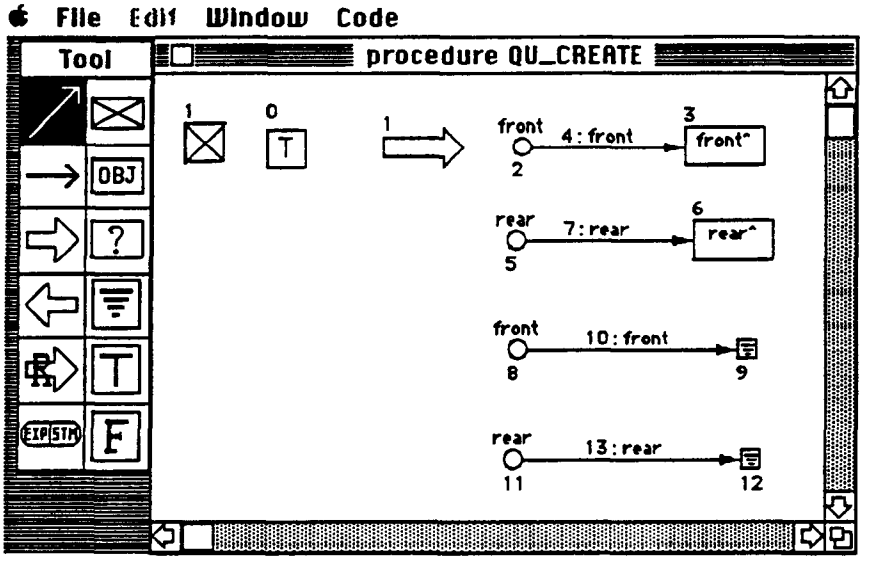


Figure F.2.3: QU_CREATE for the queue example.

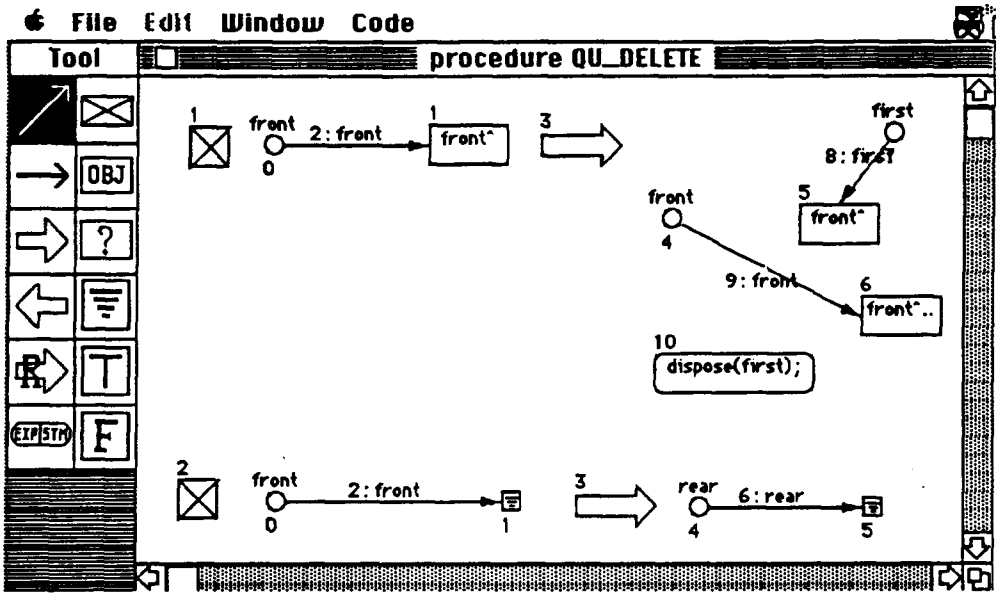


Figure F.2.4: QU_DELETE for the queue example.

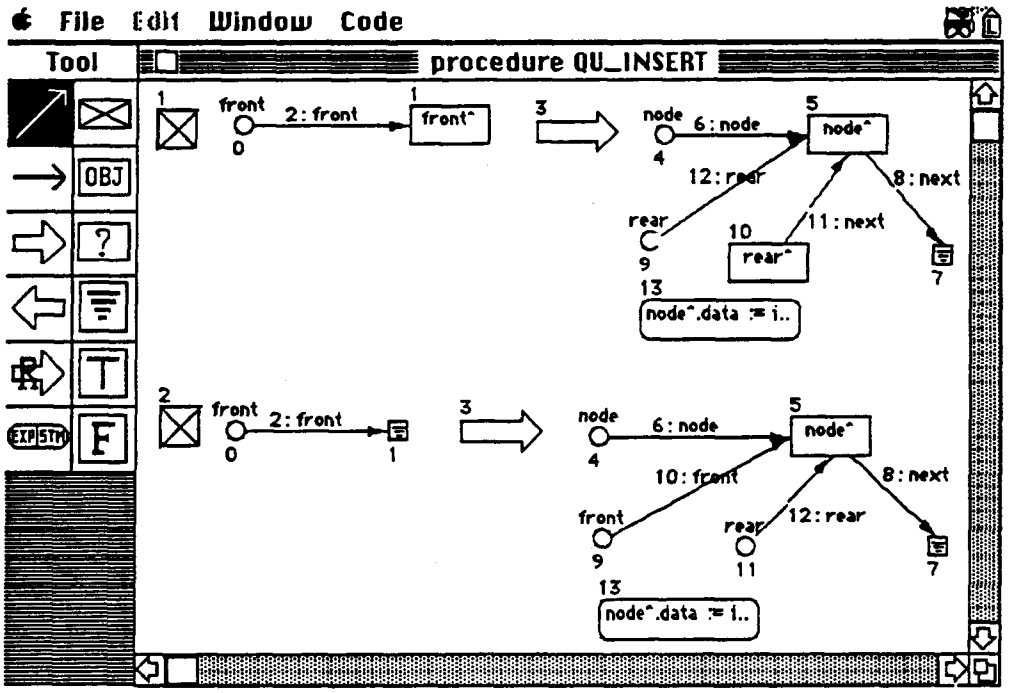


Figure F.2.5: QU_INSERT for the queue example.

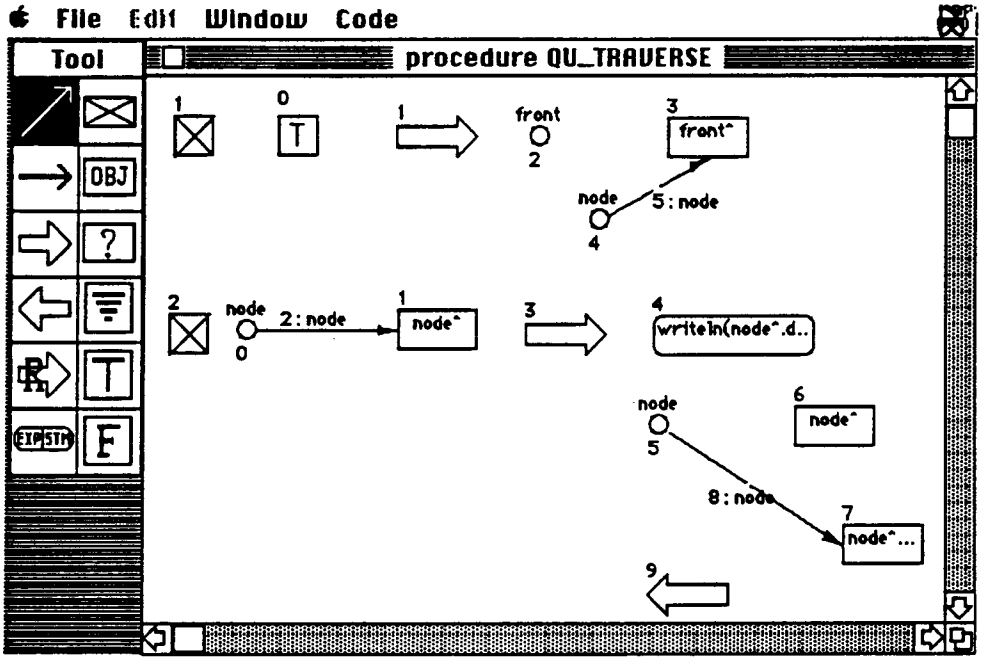


Figure F.2.6: QU_TRAVERSE for the queue example.

```
{ This unit implements a queue of integers.}
{ It inserts new elements at the end,}
{ and deletes elements from the front.}

unit QUEUE;
interface
 procedure QU_CREATE;
 procedure QU_DELETE;
 procedure QU_INSERT (item: integer);
 procedure QU_TRAVERSE;

implementation
 type
 QU_queue = ^QU_node;
 QU_node = record
   data: integer;
   next: QU_queue;
  end;

 var
  front: QU_queue;
  rear: QU_queue;
 procedure QU_CREATE;
 begin {procedure QU_CREATE}
  new(front);
  new(rear);
  front := nil;
  rear := nil;
 end; {procedure QU_CREATE}
 procedure QU_DELETE;
  var
   first: QU_queue;
 begin {procedure QU_DELETE}
  if (front <> nil) then
   begin
    first := front;
    front := front^.next;
    dispose(first);
  end;
  if (front = nil) then
   begin
    rear := nil;
  end;
 end; {procedure QU_DELETE}
 procedure QU_INSERT;
  var
   node: QU_queue;
 begin {procedure QU_INSERT}
  if (front <> nil) then
   begin
   new(node);
    node^.next := nil;
   rear^.next := node;
   rear := node;
   node^.data := item;
  end;
  if (front = nil) then
   begin
   new(node);
    node^.next := nil;
    front := node;
```

```
   rear := node;
   node^.data := item;
  end;
 end; {procedure QU_INSERT}
 procedure QU_TRAVERSE;
  var
   node: QU_queue;
 begin {procedure QU_TRAVERSE}
  node := front;
  while (node <> nil) do
   begin
    writeln(node^.data);
    node := node^.next;
   end;
 end; {procedure QU_TRAVERSE}
end.
```

Figure F.2.7: The generated Pascal code for the queue example.

```
program queue;
 uses
  QUEUE;
begin
 showText;
 QU_CREATE;
 QU_INSERT(5);
 QU_INSERT(17);
 QU_INSERT(4);
 QU_INSERT(35);
 QU_INSERT(11);
 QU_INSERT(99);
 writeln('The elements in the queue are:');
 QU_TRAVERSE;
 writeln;
 QU_DELETE;
 writeln('The elements in the queue after deleting the front are:');
 QU_TRAVERSE;
end.
```

Figure F.2.8: The test driver for the queue example.

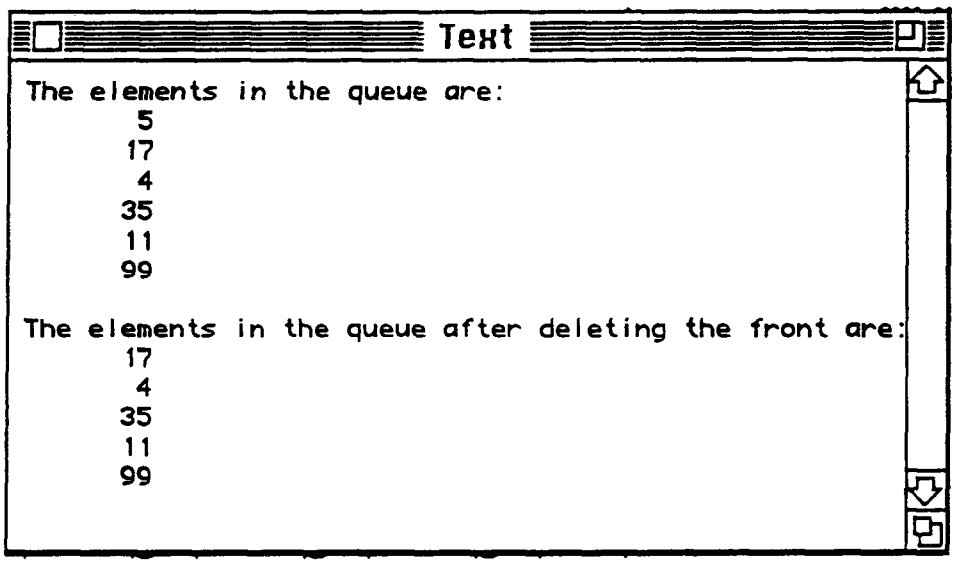


Figure F.2.9: The test results for the queue example.

3) Single Linked List with Sorting Operation

This example implements a single linked list of integers. It includes an operation to sort a single linked list as follows. It creates a new list called "sortedList", then it performs the following three steps until "list" is empty. It finds the largest element in "list", inserts that element in "sortedList", and deletes the same element from "list". This example has nine operations: SL_SORT sorts a single linked list of integers, SL_CREATE creates an instance of the list, SL_INSERT inserts an integer into a list, SL_DELETE deletes an integer from a list, SL_FINDMAX finds the maximum integer in a list, SL_delete_aux is called by SL_DELETE to delete an integer within a list (not the first one), SL_insert_aux is called by SL_SORT to insert an integer into "sortedList", and SL_max compares two integers and finds the larger one. The specifications for this example, the generated code, the test driver, and the test results are shown in Figures F.3.1-F.3.13.

File Edit Window Code

SINGLE_LINKED_LIST

```
{ This unit implements a linked list of integers.}
{ It includes operations such as sort, create}
{ insert, delete, traverse, and find maximum.}

unit SINGLE_LINKED_LIST;
interface
procedure SL_SORT;
procedure SL_CREATE;
procedure SL_INSERT(item : integer);
procedure SL_DELETE(item : integer);
procedure SL_TRAVERSE;
procedure SL_FINDMAX(var max : integer);
procedure SL_delete_aux(item : integer);
procedure SL_insert_aux(item : integer);
procedure SL_max(var max : integer; item : integer);

implementation
type
     SL_list = ^SL_node;
     SL_node = record
               data : integer;
               next : SL_list;
               end;
end.
```

Figure F.3.1: Interface for the single linked list example.

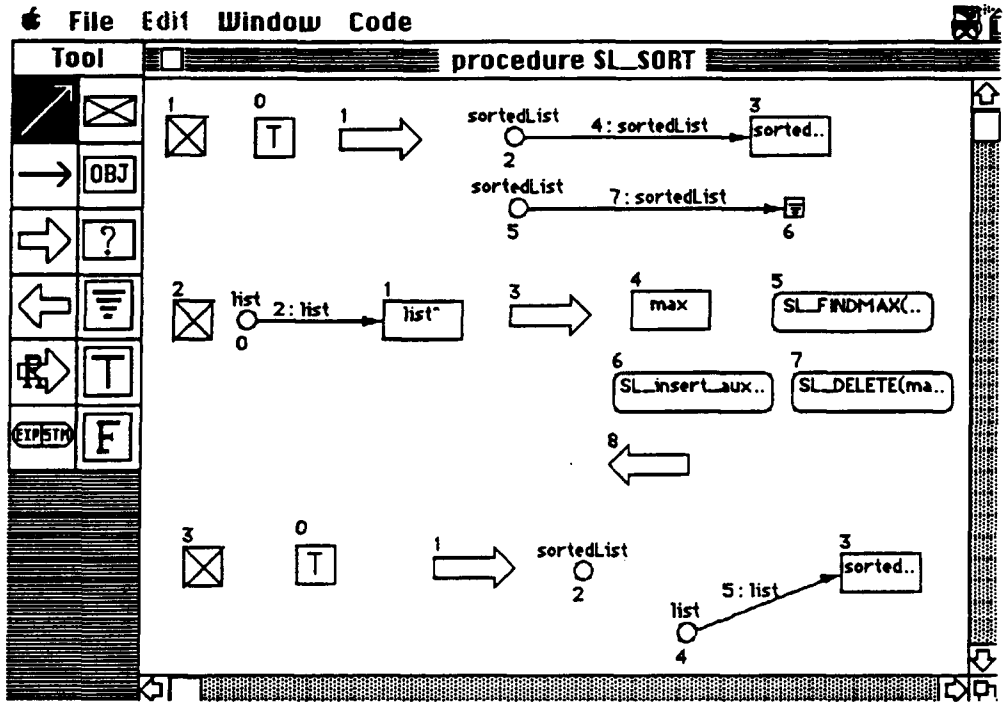


Figure F.3.2: SL_SORT operation for the single linked list example.

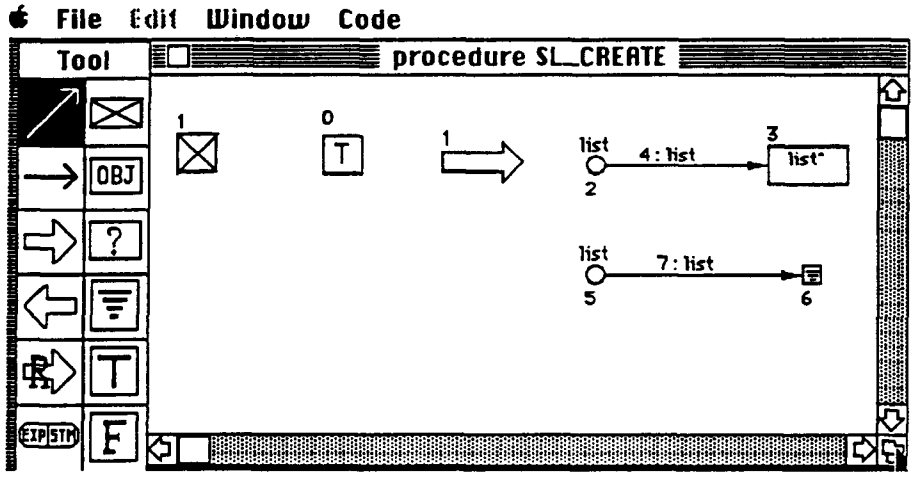


Figure F.3.3: SL_CREATE operation for the single linked list example.

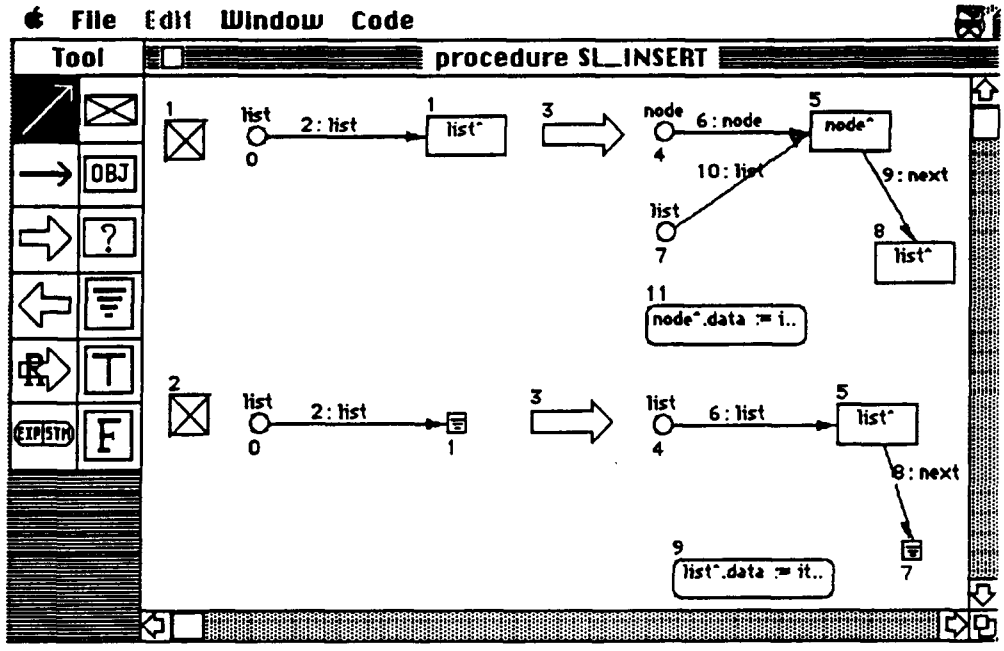


Figure F.3.4: SL_INSERT operation for the single linked list example.

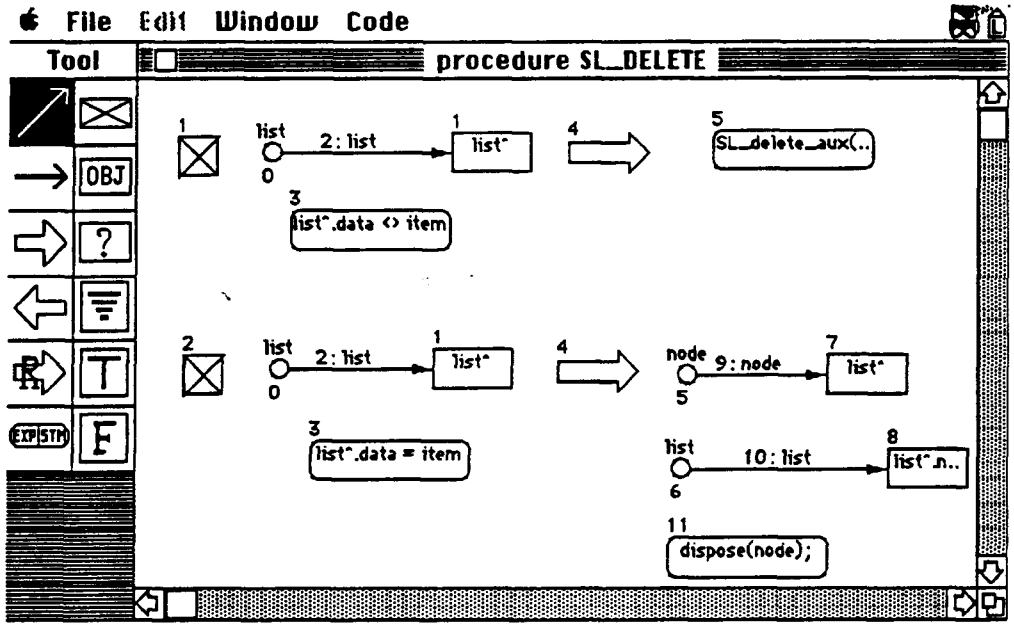


Figure F.3.5: SL_DELETE operation for the single linked list example.

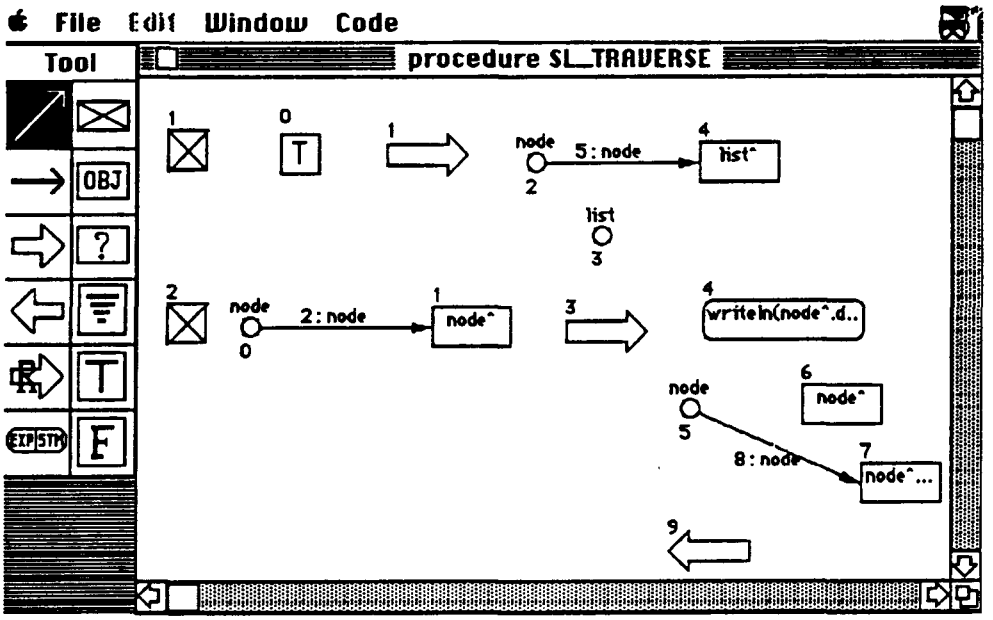


Figure F.3.6: SL_TRAVERSE operation for the single linked list example.

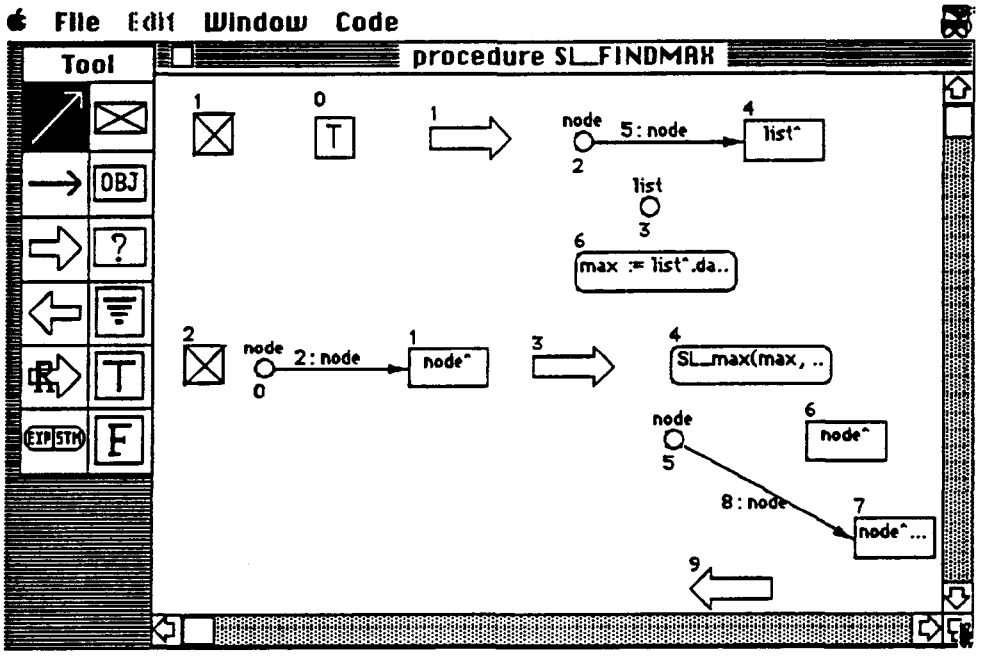


Figure F.3.7: SL_FINDMAX operation for the single linked list example.

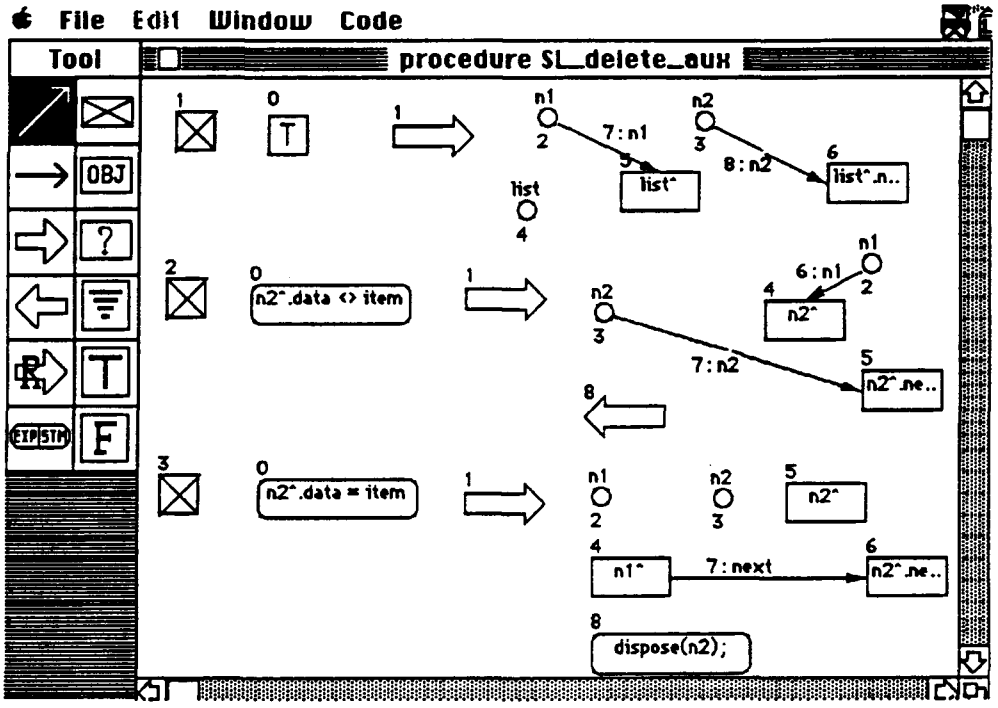


Figure F.3.8: SL_delete_aux operation for the single linked list example.

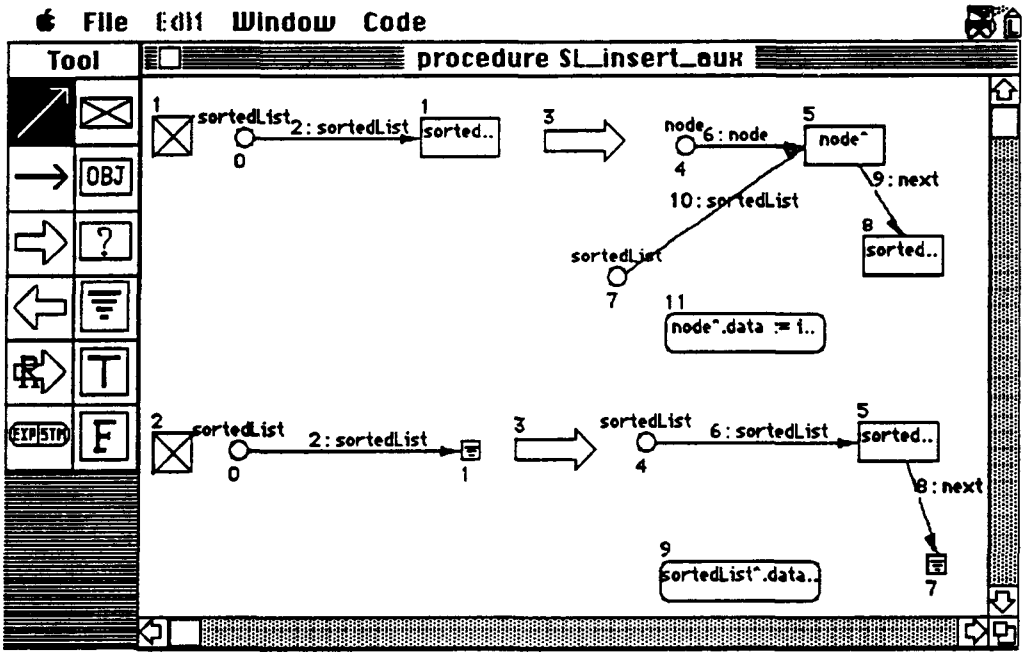


Figure F.3.9: SL_insert_aux operation for the single linked list example.

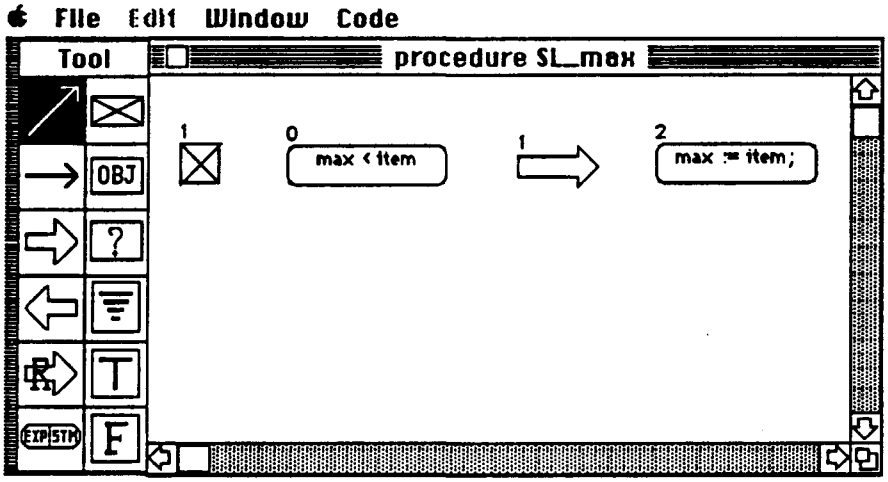


Figure F.3.10: SL_max operation for the single linked list example.

```
{ This unit implements a linked list of integers.}
{ It includes operations such as sort, create}
{ insert, delete, traverse, and find maximum.}

unit SINGLE_LINKED_LIST;

interface
 procedure SL_SORT;
 procedure SL_CREATE;
 procedure SL_INSERT (item: integer);
 procedure SL_DELETE (item: integer);
 procedure SL_TRAVERSE;
 procedure SL_FINDMAX (var max: integer);
 procedure SL_delete_aux (item: integer);
 procedure SL_insert_aux (item: integer);
 procedure SL_max (var max: integer; item: integer);

implementation
 type
  SL_list = ^SL_node;
  SL_node = record
    data: integer;
    next: SL_list;
   end;

 var
  sortedList: SL_list;
  list: SL_list;
procedure SL_SORT;
 var
  max: INTEGER;
begin {procedure SL_SORT}
 new(sortedList);
 sortedList := nil;
 while (list <> nil) do
  begin
  SL_FINDMAX(max);
   SL_insert_aux(max);
  SL_DELETE(max);
 end;
 list := sortedList;
end; {procedure SL_SORT}
procedure SL_CREATE;
begin {procedure SL_CREATE}
 new(list);
 list := nil;
end; {procedure SL_CREATE}
procedure SL_INSERT;
 var
  node: SL_list;
begin {procedure SL_INSERT}
 if (list <> nil) then
  begin
  new(node);
   node^.next := list;
   list := node;
   node^.data := item;
  end;
 if (list = nil) then
  begin
   new(list);
```

```
    list^.next := nil;
    list^.data := item;
   end;
 end; {procedure SL_INSERT}
 procedure SL_DELETE;
  var
   node: SL_list;
 begin {procedure SL_DELETE}
  if (list <> nil) and (list^.data <> item) then
   begin
    SL_delete_aux(item);
   end;
  if (list <> nil) and (list^.data = item) then
   begin
    node := list;
    list := list^.next;
    dispose(node);
   end;
 end; {procedure SL_DELETE}
 procedure SL_TRAVERSE;
  var
   node: SL_list;
 begin {procedure SL_TRAVERSE}
  node := list;
  while (node <> nil) do
   begin
    writeln(node^.data);
    node := node^.next;
   end;
 end; {procedure SL_TRAVERSE}
 procedure SL_FINDMAX;
  var
   node: SL_list;
 begin {procedure SL_FINDMAX}
  node := list;
  max := list^.data;
  while (node <> nil) do
   begin
    SL_max(max, node^.data);
    node := node^.next;
   end;
 end; {procedure SL_FINDMAX}
 procedure SL_delete_aux;
  var
   n1: SL_list;
   n2: SL_list;
 begin {procedure SL_delete_aux}
  n1 := list;
  n2 := list^.next;
  while (n2^.data <> item) do
   begin
    n1 := n2;
    n2 := n2^.next;
   end;
  if (n2^.data = item) then
   begin
    n1^.next := n2^.next;
    dispose(n2);
   end;
 end; {procedure SL_delete_aux}
 procedure SL_insert_aux;
```

```
 var
  node: SL_list;
 begin {procedure SL_insert_aux}
  if (sortedList <> nil) then
   begin
   new(node);
    node^.next := sortedList;
    sortedList := node;
    node^.data := item;
  end;
  if (sortedList = nil) then
   begin
    new(sortedList);
     sortedList^.next := nil;
     sortedList^.data := item;
  end;
 end; {procedure SL_insert_aux}
 procedure SL_max;
 begin {procedure SL_max}
  if (max < item) then
   begin
    max := item;
  end;
 end; {procedure SL_max}
end.
```

Figure F.3.11: The generated Pascal code for the single linked list example.

```
program singleList;
 uses
  SINGLE_LINKED_LIST;
begin
 showText;
 SL_CREATE;
 SL_INSERT(2);
 SL_INSERT(5);
 SL_INSERT(3);
 SL_INSERT(7);
 SL_INSERT(1);
 writeln('The elements before sorting are:');
 SL_TRAVERSE;

 SL_SORT;
 writeln;
 writeln('The elements after sorting are:');
 SL_TRAVERSE;
end.
```

Figure F.3.12: The test driver for the single linked list example.

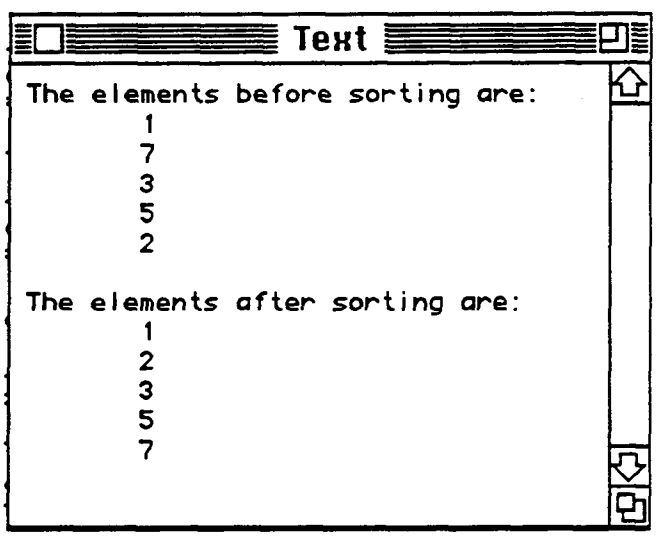


Figure F.3.13: The test results for the single linked list example.

4) Double Linked List

This example implements a circular double linked list (ring) of integers. An empty ring is defined as a ring with a single dummy element linked to itself (Grogono, 1980), as shown in Figure F.4.1. This example has four operations: DL_CREATE to create an instance of the double linked list, DL_DELETE to delete an integer from the ring, DL_INSERT to insert an integer at the front of the ring, and DL_TRAVERSE to print the integers in the ring.The specifications of this module, the generated code, the test driver, and the test results are shown in Figures F.4.2-F.4.9.

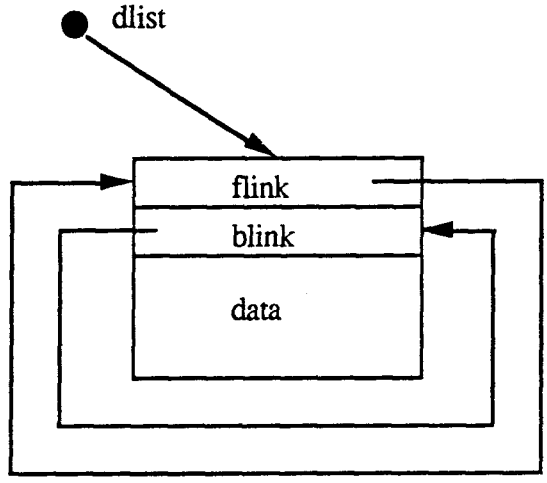


Figure F.4.1: An empty double linked list with a dummy element.

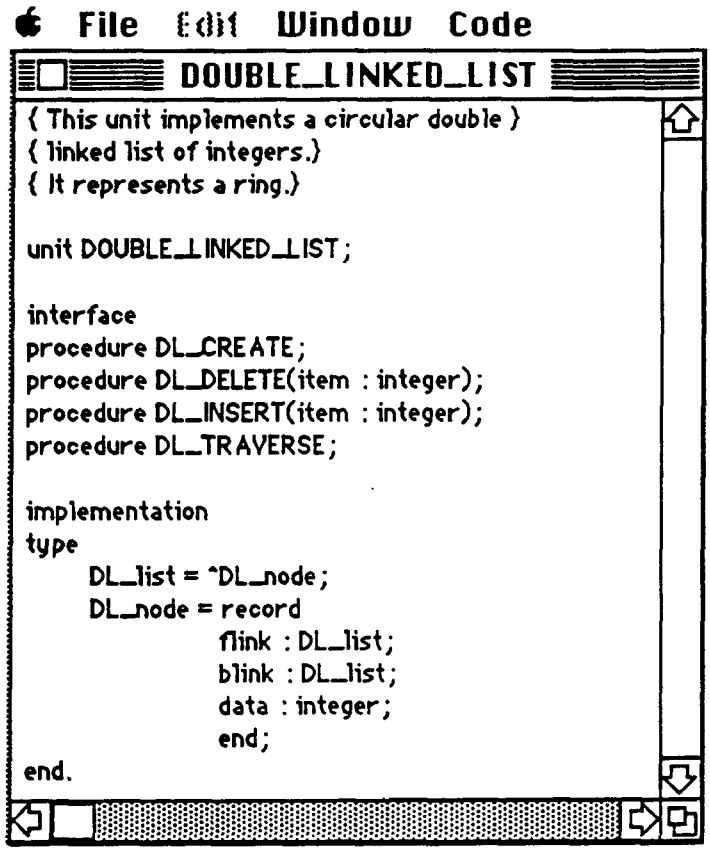


Figure F.4.2: Interface for the double linked list example.

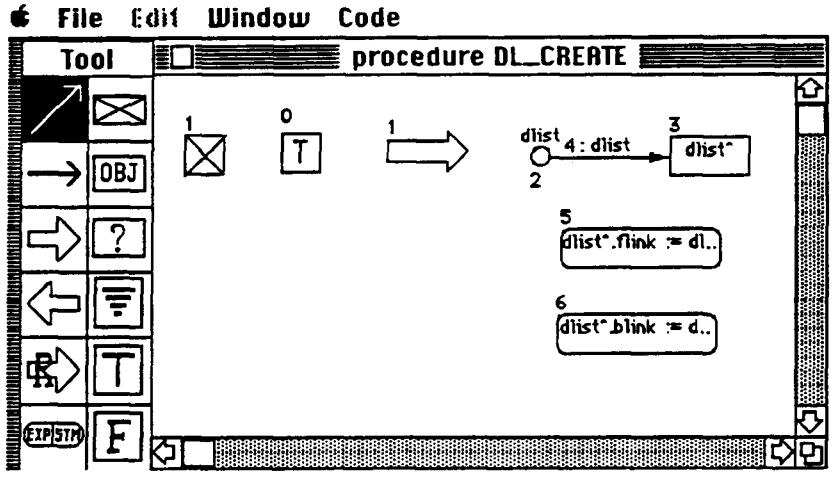


Figure F.4.3: DL_CREATE for the double linked list example.

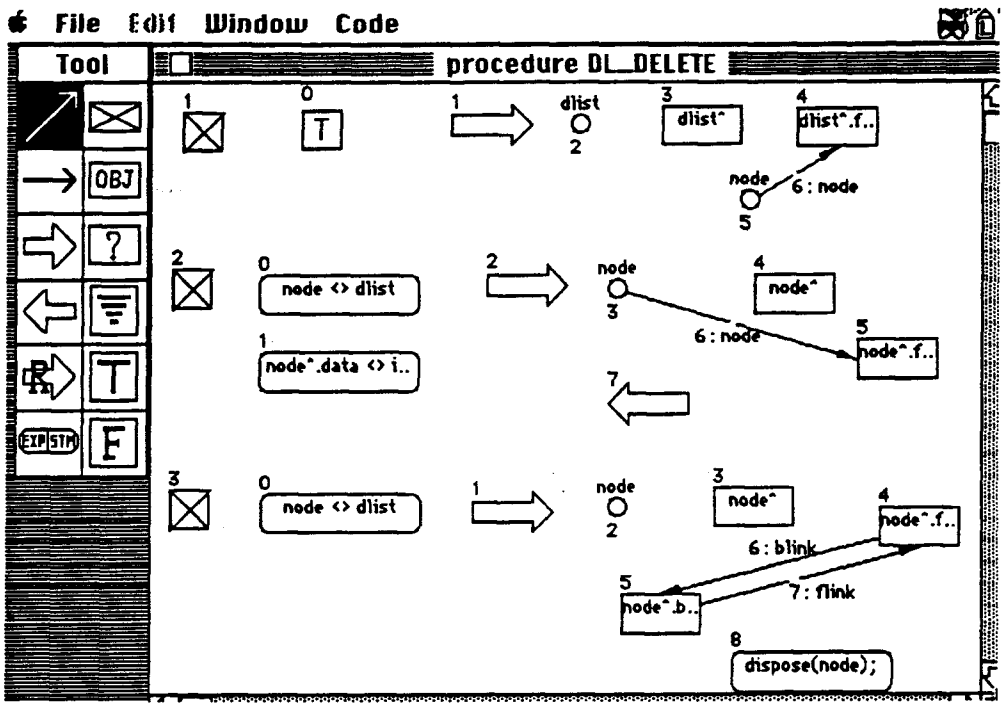


Figure F.4.4: DL_DELETE for the double linked list example.

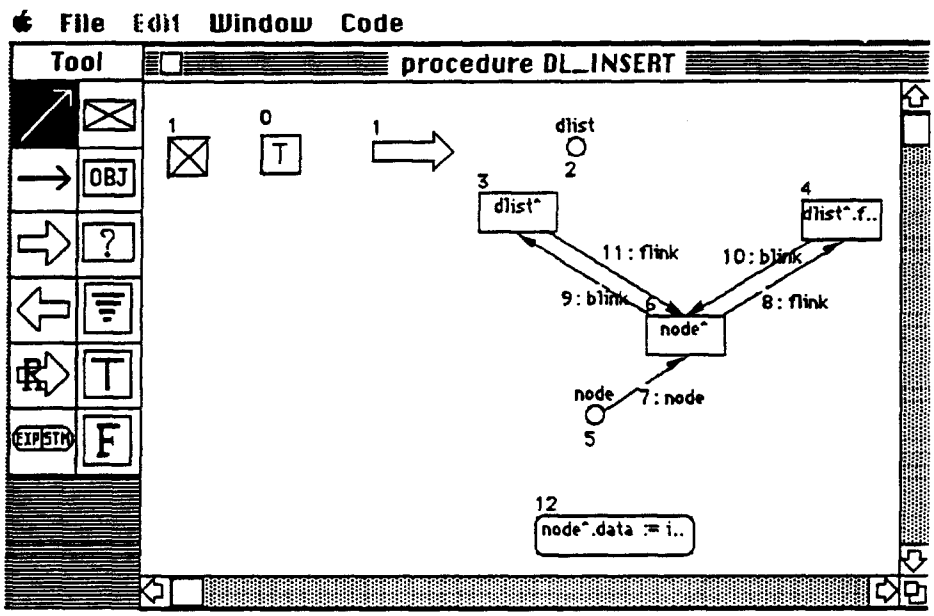


Figure F.4.5: DL_INSERT for the double linked list example.

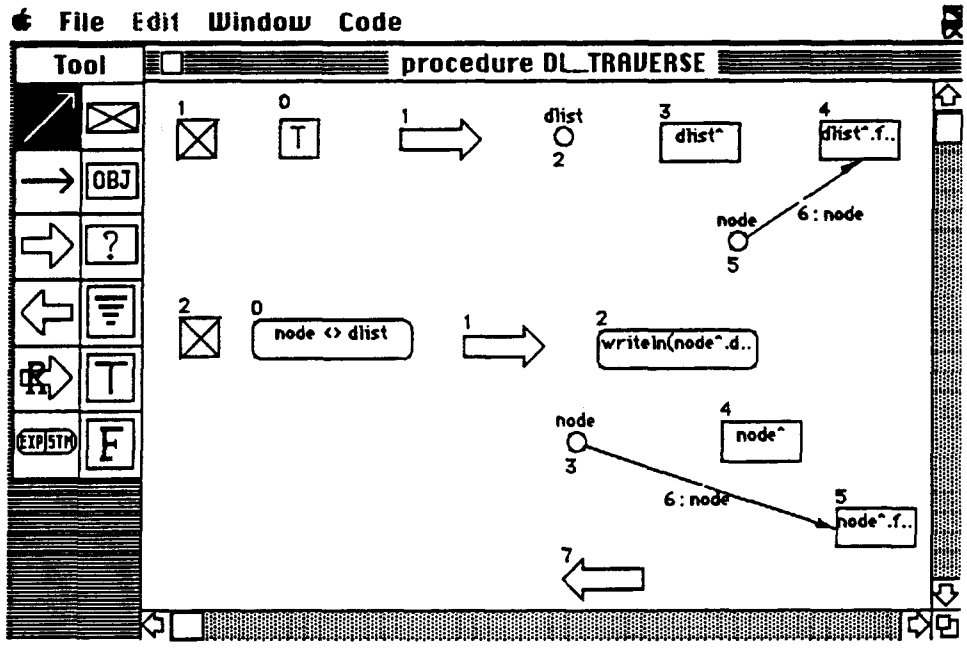


Figure F.4.6: DL_TRAVERSE for the double linked list example.

```
{ This unit implements a circular double }
{ linked list of integers.}
{ It represents a ring.}

unit DOUBLE_LINKED_LIST;

interface
 procedure DL_CREATE;
 procedure DL_DELETE (item: integer);
 procedure DL_INSERT (item: integer);
 procedure DL_TRAVERSE;

implementation
 type
  DL_list = ^DL_node;
  DL_node = record
    flink: DL_list;
    blink: DL_list;
    data: integer;
   end;

 var
  dlist: DL_list;
 procedure DL_CREATE;
 begin {procedure DL_CREATE}
  new(dlist);
  dlist^.flink := dlist;
  dlist^.blink := dlist;
 end; {procedure DL_CREATE}
 procedure DL_DELETE;
  var
   node: DL_list;
 begin {procedure DL_DELETE}
  node := dlist^.flink;
  while (node <> dlist) and (node^.data <> item) do
   begin
    node := node^.flink;
   end;
  if (node <> dlist) then
   begin
    node^.flink^.blink := node^.blink;
    node^.blink^.flink := node^.flink;
    dispose(node);
   end;
 end; {procedure DL_DELETE}
 procedure DL_INSERT;
  var
   node: DL_list;
 begin {procedure DL_INSERT}
  new(node);
  node^.flink := dlist^.flink;
  node^.blink := dlist;
  dlist^.flink^.blink := node;
  dlist^.flink := node;
  node^.data := item;
 end; {procedure DL_INSERT}
 procedure DL_TRAVERSE;
  var
   node: DL_list;
 begin {procedure DL_TRAVERSE}
  node := dlist^.flink;
```

```
  while (node <> dlist) do
   begin
    writeln(node^.data);
    node := node^.flink;
   end;
 end; {procedure DL_TRAVERSE}
end.
```

Figure F.4.7: The generated Pascal code for the double linked list example.

```
program doublelinkedlist;
 uses
  DOUBLE_LINKED_LIST;
begin
 showText;
 DL_CREATE;
 DL_INSERT(5);
 DL_INSERT(20);
 DL_INSERT(12);
 DL_INSERT(2);
 DL_INSERT(37);

 writeln('The elements in the double linked list are:');
 DL_TRAVERSE;
 writeln;
 DL_DELETE(5);
 writeln('The elements after deleting 5 are:');
 DL_TRAVERSE;
 DL_DELETE(12);
 writeln('The elements after deleting 12 are:');
 DL_TRAVERSE;
 DL_DELETE(37);
 writeln('The elements after deleting 37 are:');
 DL_TRAVERSE;
end.
```

Figure F.4.8: The test driver for the double linked list example.

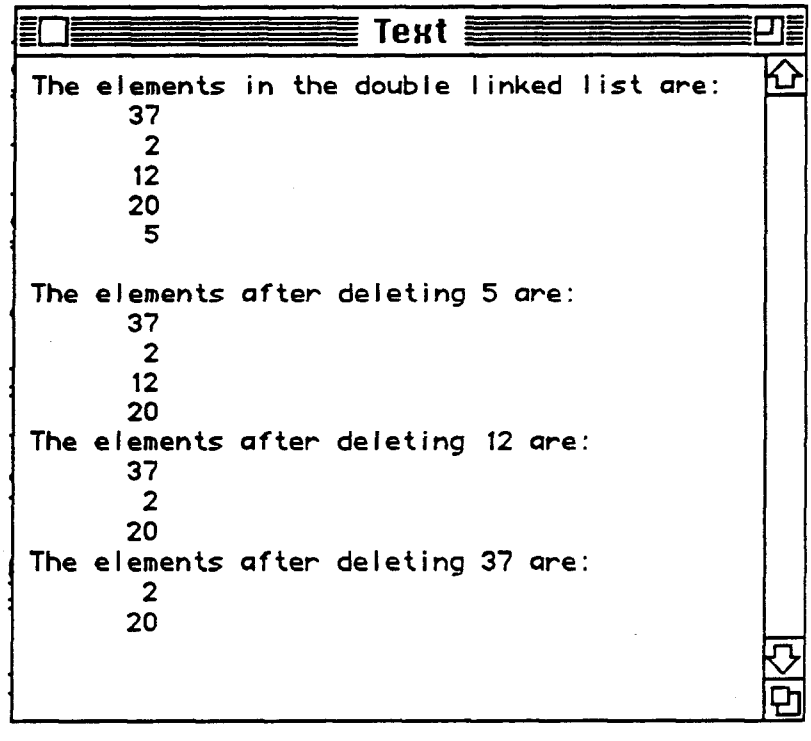


Figure F.4.9: The test results for the double linked list example.

5) Binary Tree

This example implements a binary tree module with four operations: BT_CREATE, BT_TRAVERSE, BT_INSERT, and BT_ISEMPTY. BT_CREATE creates an instance of the binary tree, BT_TRAVERSE implements a preorder tree traversal, BT_INSERT inserts an integer into the binary tree, and BT_ISEMPTY returns "true" if the tree is empty (nil), and returns "false" if the tree is not empty. These operations, along with the interface part, the generated Pascal code, the test driver, and the test results are shown in Figures F.5.1-F.5.8. This BINARY_TREE module uses the AUXILIARY_TREE module to accomplish a recursive implementation of BT_TRAVERSE and BT_INSERT operations for a non-empty tree. The specifications and the generated code for the later module are shown in Figures F5.9-F5.12. We don't show a separate test case for the AUXILIARY_TREE module because the test driver in Figure F5.7 tests both modules.

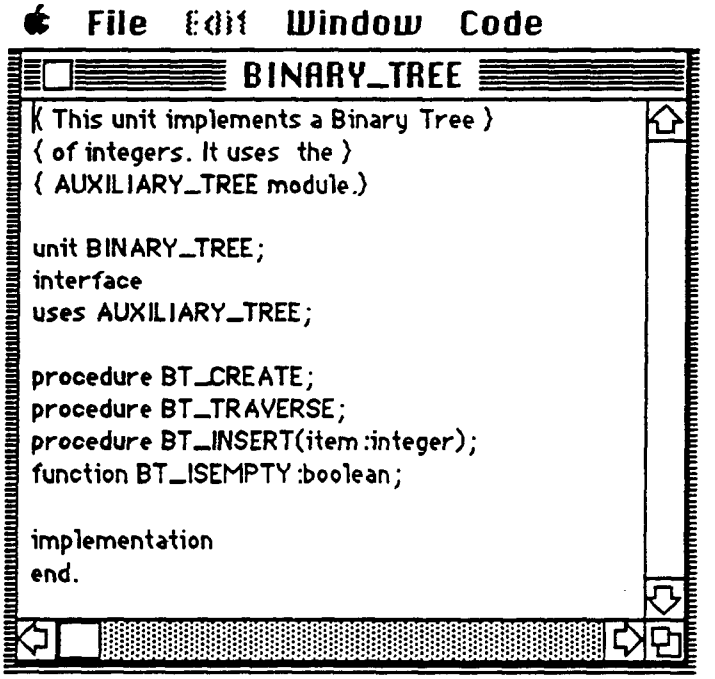


Figure F.5.1: Interface for the binary tree example.

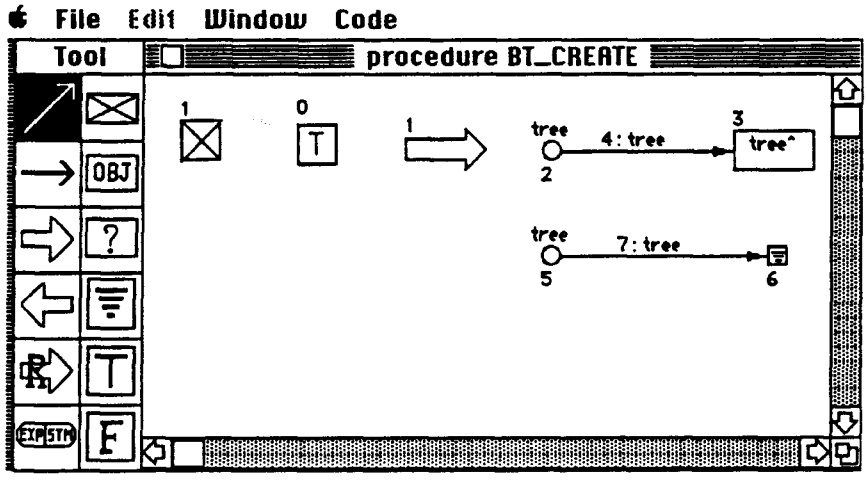


Figure F.5.2: BT_CREATE operation for the binary tree example.

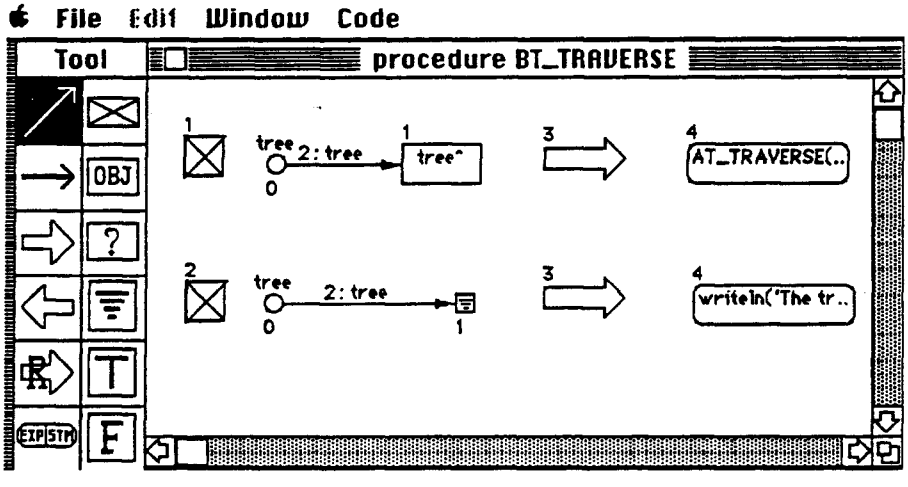


Figure F.5.3: BT_TRAVERSE operation for the binary tree example.

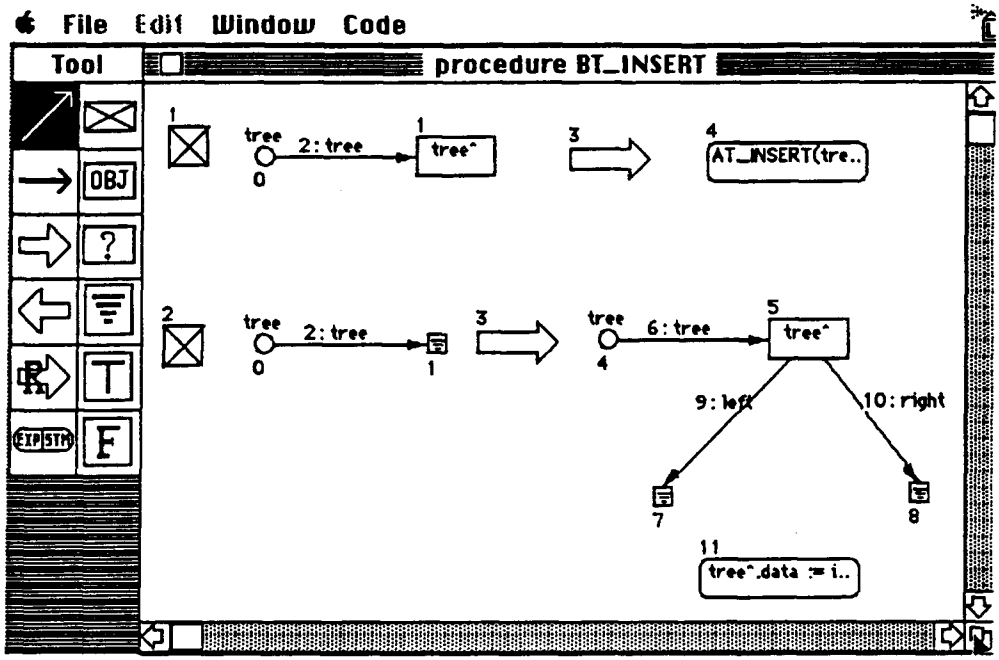


Figure F.5.4: BT_INSERT operation for the binary tree example.

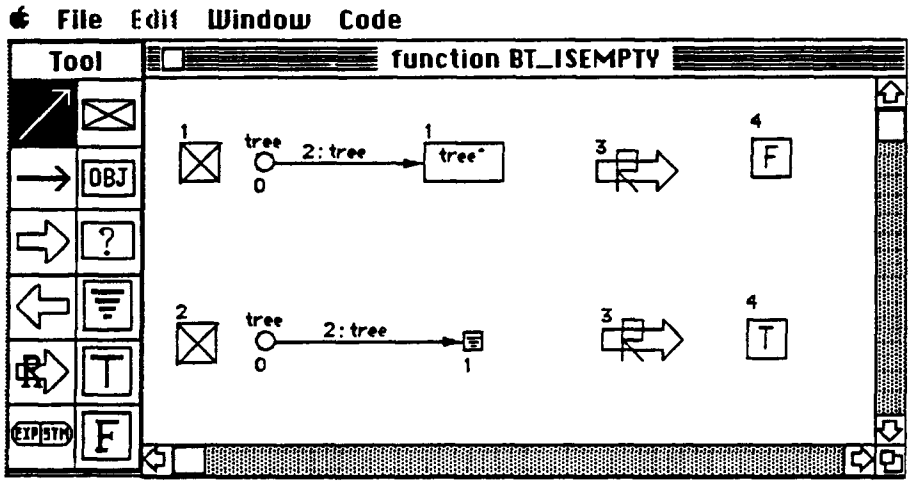


Figure F.5.5: BT_ISEMPTY operation for the binary tree example.

```
{ This unit implements a Binary Tree }
{ of integers. It uses the }
{ AUXILIARY_TREE module.}

unit BINARY_TREE;
interface
 uses
  AUXILIARY_TREE;

 procedure BT_CREATE;
 procedure BT_TRAVERSE;
 procedure BT_INSERT (item: integer);
 function BT_ISEMPTY: boolean;

implementation

 var
  tree: AT_tree;
 procedure BT_CREATE;
 begin {procedure BT_CREATE}
  new(tree);
  tree := nil;
 end; {procedure BT_CREATE}
 procedure BT_TRAVERSE;
 begin {procedure BT_TRAVERSE}
  if (tree <> nil) then
   begin
    AT_TRAVERSE(tree);
   end;
  if (tree = nil) then
   begin
    writeln('The tree is empty');
   end;
 end; {procedure BT_TRAVERSE}
 procedure BT_INSERT;
 begin {procedure BT_INSERT}
  if (tree <> nil) then
   begin
    AT_INSERT(tree, item);
   end;
  if (tree = nil) then
   begin
    new(tree);
    tree^.left := nil;
    tree^.right := nil;
    tree^.data := item;
   end;
 end; {procedure BT_INSERT}
 function BT_ISEMPTY;
 begin {function BT_ISEMPTY}
  if (tree <> nil) then
   begin
    BT_ISEMPTY := false;
   end;
  if (tree = nil) then
   begin
    BT_ISEMPTY := true;
   end;
 end; {function BT_ISEMPTY}
end.
```

Figure F.5.6: The generated Pascal code for the binary tree example.

```
program binarytree;
 uses
  BINARY_TREE;
begin
 showText;
 BT_CREATE;
 BT_INSERT(10);
 BT_INSERT(8);
 BT_INSERT(25);
 BT_INSERT(4);
 BT_INSERT(9);
 BT_INSERT(20);
 BT_INSERT(30);
 writeln('The preorder traversal of the elements in the tree are:');
 BT_TRAVERSE;
 writeln;
 writeln('Is the tree empty? ', BT_ISEMPTY);
end.
```

Figure F.5.7: The test driver for the binary tree example.

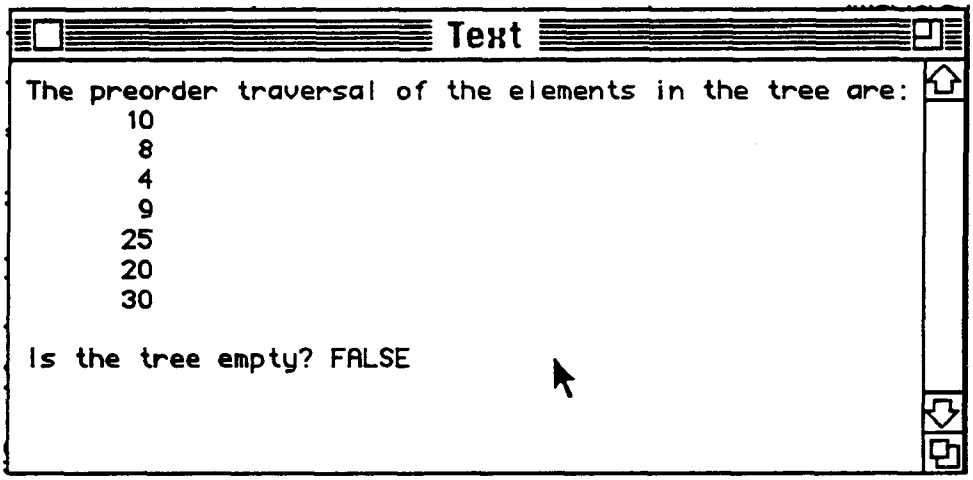


Figure F.5.8: The test results for the binary tree example.

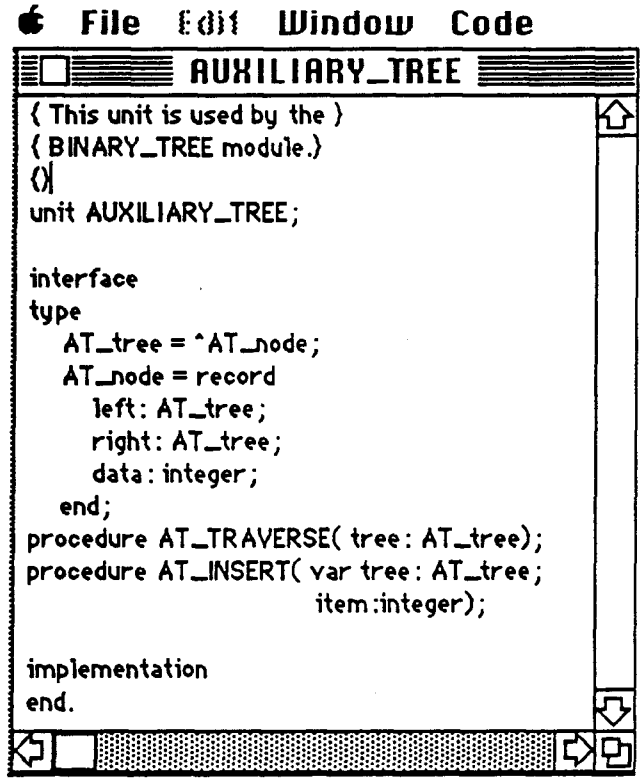


Figure F.5.9: Interface for the AUXILIARY_TREE module.

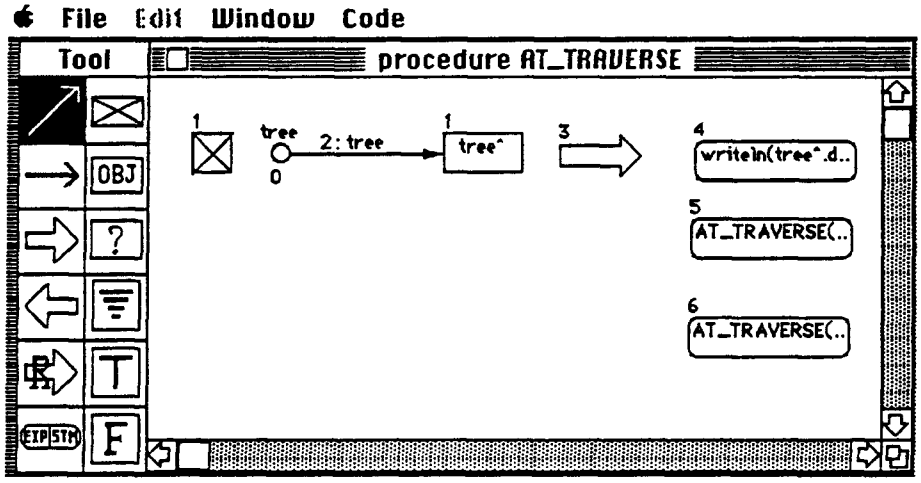


Figure F.5.10: AT_TRAVERSE operation for the AUXILIARY_TREE module.

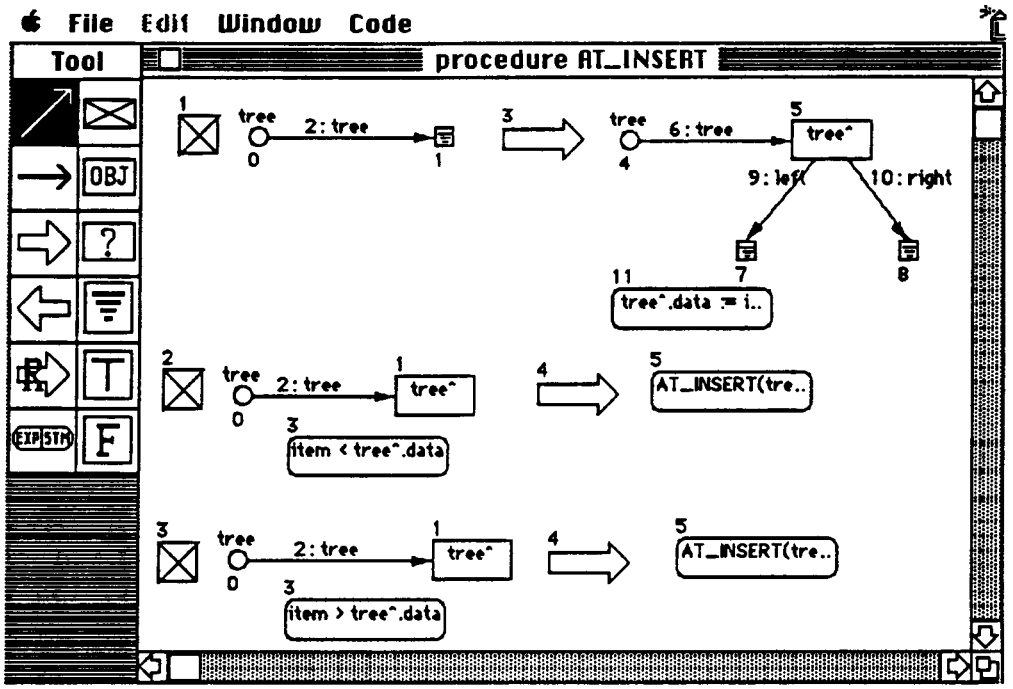


Figure F.5.11: AT_INSERT for the AUXILIARY_TREE module.

```
{ This unit is used by the }
{ BINARY_TREE module.}
{}

unit AUXILIARY_TREE;

interface
 type
  AT_tree = ^AT_node;
  AT_node = record
    left: AT_tree;
    right: AT_tree;
    data: integer;
   end;
 procedure AT_TRAVERSE (tree: AT_tree);
 procedure AT_INSERT (var tree: AT_tree; item: integer);

implementation

 procedure AT_TRAVERSE;
 begin {procedure AT_TRAVERSE}
  if (tree <> nil) then
   begin
    writeln(tree^.data);
   AT_TRAVERSE(tree^.left);
   AT_TRAVERSE(tree^.right);
   end;
 end; {procedure AT_TRAVERSE}
 procedure AT_INSERT;
 begin {procedure AT_INSERT}
  if (tree = nil) then
   begin
   new(tree);
    tree^.left := nil;
    tree^.right := nil;
   tree^.data := item;
  end;
  if (tree <> nil) and (item < tree^.data) then
   begin
   AT_INSERT(tree^.left, item);
  end;
  if (tree <> nil) and (item > tree^.data) then
   begin
   AT_INSERT(tree^.right, item);
  end;
 end; {procedure AT_INSERT}
end.
```

Figure F.5.12: The generated Pascal code for the AUXILIARY_TREE module.